Universidad Carlos III de Madrid
Escuela Politécnica Superior

DESARROLLO E IMPLEMENTACIÓN DE UN CLIENTE PARA EL ENTORNO VIRTUAL AI-LIVE

Proyecto Fin de Carrera
Ingeniería Técnica en Informática de Gestión

Autor: Alberto Uceda Blanco
Tutor: Daniel Pérez Pinillos
Directora: Susana Fernández Arregui

Agradecimientos

A Javier Escudero, por trabajar con él codo con codo mucho tiempo y ser además una de las mejores personas que te puedes echar de amigo.

Al resto de compañeros que han pasado por el proyecto (Marta, Iván, Pilar, Miguel) y que han contribuido a desarrollar y ampliar esta aplicación.

A los tutores por darme a conocer este proyecto, colaborar en su desarrollo y formar parte de él.

A mis padres y mi hermana por los constantes ánimos y su apoyo.

A Kike, María, Ángela, Alberto, por pasar días y días con ellos en la Universidad mientras estudiábamos o desarrollaba el proyecto y aguantar mis charlas.

A mi grupo de Informatilocos por todos los momentos compartidos durante la carrera.

Gracias a todos.

Índice de contenido

Índice de figuras

1 INTRODUCCIÓN

Este apartado del documento muestra un resumen sobre el ámbito en el que se ha desarrollado este Proyecto Fin de Carrera, su objetivo y la estructuración de la memoria.

1.1 Entorno del Proyecto Fin de Carrera

El presente proyecto se enmarca en el área de Inteligencia Artificial. AI-LIVE es un juego de simulación social, basado en el videojuego Los Sims, que fue creado con el objetivo de simular situaciones del mundo real. En el juego, los personajes, llamados actores, toman decisiones para realizar determinadas acciones dentro del entorno donde se encuentran, que vienen influidas por las características de los personajes y sus necesidades.

El juego está basado en una arquitectura cliente-servidor. Los actores anteriormente mencionados, se implementan como clientes, que contienen el código necesario para permitir al actor tomar sus decisiones y enviárselas al servidor. El servidor es el encargado de actualizar el estado del juego según las acciones realizadas por los clientes y de asignar los turnos a éstos.

Actualmente en AI-LIVE existen cuatro tipos de clientes: el cliente CLIPS, que decide qué acción quiere realizar utilizando un Sistema basado en reglas; el cliente manual CLIPS, que es igual que el anterior salvo que es el usuario el encargado de elegir la acción que desea realizar; el cliente Prodigy (consistente en un planificador de tareas), el cual trata de conseguir un objetivo dada su descripción, un estado y un conjunto de acciones; y el cliente GUI, que es el encargado de mostrar la interfaz gráfica 3D que representa los estados del juego.

Además, el servidor consta de un motor emocional, encargado de controlar las emociones, los gustos por objetos u otros elementos, y las relaciones entre actores cuando uno inicia una comunicación verbal con otro.

1.2 Cometido del Proyecto Fin de Carrera

Se pretende desarrollar e implementar un cliente para el juego AI-LIVE, que añada nuevas funcionalidades a las ya existentes en versiones anteriores, así como realizar modificaciones y mejoras. La implementación del cliente se realiza en paralelo con el desarrollo del servidor, asignado en otro Proyecto Fin de Carrera, coordinándose ambos para su correcta compilación y funcionamiento.

Con la implementación de este cliente, se pretende aumentar la complejidad del sistema en el que los actores puedan realizar más acciones, aumentando así las

posibles decisiones a tomar y obteniendo por tanto diferentes ejecuciones por cada vez que se lance el juego.

También se va a realizar el acoplamiento del Modelo Emocional, realizado anteriormente por Marta Jiménez [2008], al sistema implementado, para conseguir que las emociones también tengan un importante papel en el universo AI-LIVE, afectando a las decisiones tomadas por los actores, así como las relaciones entre los propios actores.

1.3 Estructura del documento

Capítulo 2. Estado del arte: En primer lugar se hará un repaso a la evolución de los videojuegos, explicando posteriormente la necesidad de la existencia de la Inteligencia Artificial en ellos. Se comentarán las técnicas de Inteligencia Artificial conocidas y utilizadas, y se mostrarán varios ejemplos de videojuegos que aplican estas técnicas observando para qué se utilizan.

Capítulo 3. Objetivos: se describen las diferentes metas que se desean alcanzar con la realización del proyecto.

Capítulo 4. Trabajo realizado: se describe el trabajo realizado en la aplicación AI-LIVE. Detalla la arquitectura de la aplicación, el modelo de conocimiento actual, la descripción del cliente Inteligencia Artificial, que es el que se trata, y la documentación de las diversas mejoras para conseguir el objetivo del proyecto.

Capítulo 5. Pruebas y Resultados: en este capítulo se muestran los resultados de las diferentes pruebas y experimentos realizados que reflejen las ampliaciones realizadas.

Capítulo 6. Presupuesto: En este capítulo se detallará el presupuesto del proyecto.

Capítulo 7. Conclusiones: se realizará un resumen final sobres los avances de AI-LIVE, así como la resolución de diversos problemas surgidos durante el proceso.

Capítulo 8. Líneas futuras: este capítulo plantea diferentes ampliaciones del sistema, así como detalles a mejorar.

Capítulo 9. Bibliografía: se recogen las diferentes fuentes de información utilizadas para documentar y realizar el estado del arte y el Proyecto Fin de Carrera.

Capítulo 10. Anexos: en este apartado se recogen otros documentos elaborados, necesarios por su utilidad y como ayuda al código fuente del proyecto. Estos son el Manual de Usuario, que contiene indicaciones sobre cómo instalar, preparar y ejecutar la aplicación; el Manual de Referencia, con información detallada para conocimiento y ampliación de la aplicación; y un manual para instalar los componentes necesarios que permiten ejecutar el cliente gráfico de AI-LIVE.

2 ESTADO DE LA CUESTIÓN

2.1 Introducción

En este capítulo de la memoria pretendemos dar a conocer la evolución de los videojuegos, tanto técnicamente como en otros aspectos. También reflejaremos la tecnología que se usa en los videojuegos, centrándonos exclusivamente en la Inteligencia Artificial.

El propósito de este estado del arte es abarcar los motivos y preocupaciones que provocan que la Inteligencia Artificial empiece a cobrar importancia y protagonismo en la industria del videojuego, incluyendo y explicando tanto técnicas específicas usadas como las diversas aplicaciones para el diseño de videojuegos que utilizan estas técnicas.

La estructura de este capítulo se basará en una pequeña introducción donde se comentará la historia y evolución técnica de los videojuegos, desde sus inicios hasta la actualidad. Posteriormente veremos las necesidades que llevaron a empezar a utilizar Inteligencia Artificial en la industria del videojuego. Después entraremos en una clasificación de géneros de videojuegos basándonos en la importancia de tener un nivel humano de Inteligencia Artificial en ellos.

El siguiente punto del documento constará de una explicación de las técnicas más usadas en Inteligencia Artificial, así como otras más actuales. Estas técnicas se verán posteriormente reflejadas en diversas aplicaciones o problemas que las usan.

Finalmente se resumirán algunos videojuegos, tanto comerciales como de código abierto, donde se use Inteligencia Artificial y algunas de las técnicas que utiliza. En algunos videojuegos de código abierto no se comentará acerca de la Inteligencia Artificial, sino que se hablará de cómo colaborar o explicar el código fuente existente.

2.2 Historia de los videojuegos

Los videojuegos cuentan con una corta historia, pero a su vez intensa, con cambios tecnológicos importantes y avanzados en poco tiempo. Estos avances eran impensables cuando se diseñaron los primeros videojuegos, pero el crecimiento de la informática en general y en particular de la industria del videojuego, le ha permitido ponerse al nivel del cine, por ejemplo, en cuanto a importancia de la gente en general como entretenimiento.

Este entretenimiento comenzó por ser algo sencillo, con una idea inicial basada en el ocio hacia el público infantil, pero analizando la evolución del sector resulta paradójico hablar de un simple entretenimiento. Es cierto que la idea inicial pasa por divertir y entretener al jugador, pero las mejoras tecnológicas han permitido a los videojuegos llegar a convertirse en medios por los cuales las personas pueden, por

ejemplo, ejercitar tanto su mente como su cuerpo, llegando a cualquier tipo de público.

La evolución de los videojuegos es tal, que en apenas unas décadas han pasado de ser unos escasos polígonos a toda una recreación de grandes escenarios en tres dimensiones [Berron and Wolf, 2003], como muestran respectivamente las figuras 1 y 2 a continuación:

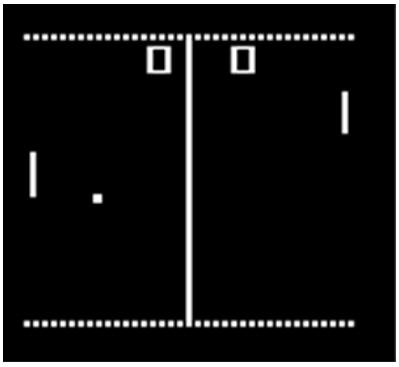

Figura 1. Pong (Primeros juegos)

Figura 2. Unreal Tournament 3 (Actualidad)

Ahora mismo es posible desde ponerse a los mandos de un avión y realizar una misión virtual por un escenario real como puede ser una ciudad metropolitana fielmente recreada; o jugar sin mandos, a través de una cámara que captura tus movimientos. Es un factor muy importante y amplio, lo que le da a la industria de los videojuegos una gran aceptación entre la gente.

Otro de los aspectos a tener en cuenta actualmente en los videojuegos, y que le proporciona variadas posibilidades de expansión es la opción multijugador, u *on-line*. Poco a poco Internet y la Red van adquiriendo protagonismo en los videojuegos, donde es una forma más de comunicarse con personas de todo el mundo [Castronova, 2005]. No se sabe con certeza qué le deparará a la industria de los videojuegos, pero no se descarta que en un futuro no muy lejano se juegue a todo a través de Internet.

Finalmente, el concepto más importante y el que nos concierne, es el de Inteligencia Artificial. Actualmente existen y se están implementando técnicas de Inteligencia Artificial de mayor complejidad que otorgan a un videojuego un alto grado de realismo, ya que acerca el comportamiento de la máquina al de una situación real. Por tanto en el siguiente punto comentaremos las necesidades que dieron lugar a empezar a implementar estas técnicas, y un breve repaso al papel de la Inteligencia Artificial en la historia de los videojuegos.

2.3 Necesidad de Inteligencia Artificial en videojuegos

La Inteligencia Artificial es una rama de la Informática que trata de resolver problemas de forma parecida a cómo los resuelven los humanos.

Los primeros fines e investigaciones de la Inteligencia Artificial estaban dirigidos al hallazgo de una técnica universal para la solución de problemas [Norvig and Russell, 2003]. Este intento ha sido abandonado, ya que era a una gran escala, y las investigaciones actuales se han dedicado al diseño y desarrollo de programas para que las máquinas sean capaces de imitar los procesos de tomas de decisiones de los humanos. Naturalmente el fin final consiste en que una máquina pueda reproducir todas las facetas de la inteligencia humana.

La Inteligencia Artificial aplicada a los videojuegos tuvo sus primeras ideas a finales de los años 50, en juegos como las *Damas*, *Ajedrez* o *Go*, en general juegos de tableros (posteriormente comentaremos algo de estos juegos), donde se pretendía que estos programas fueran capaces de jugar mediante aprendizaje (sin tener ningún conocimiento, fuera adquiriéndolo jugando partidas). Esto supuso una gran revolución, y a partir de ese momento la aplicación de técnicas de Inteligencia Artificial en los videojuegos ha tenido mayor relevancia debido a que los juegos empezaron a dejar de ser de tipo "jugador contra jugador", por lo que el usuario quiere sentir que la máquina tenga un comportamiento humano, o de un nivel de dificultad adecuado.

Esto junto con otros motivos que comentamos a continuación [Laird and van Lent, 2000], conforman la base de necesidades que han permitido a la Inteligencia Artificial conseguir un mayor protagonismo en la industria del videojuego.

1. Los desarrolladores de videojuegos empiezan a reconocer la necesidad del nivel humano de Inteligencia Artificial. Personajes humanos sintéticos empiezan a tener un papel cada vez más importante en muchos géneros de videojuegos y tienen potencial para dar lugar a nuevos géneros.

2. La industria de los videojuegos es muy competitiva, y uno de los componentes de esa competitividad es la tecnología. La Inteligencia Artificial es a menudo mencionada como la próxima tecnología que mejorará los videojuegos y determinará cuáles de ellos serán éxitos. Por eso va a tener un impacto significativo en la industria, ya que muchos juegos se comercializarán basados en la calidad de su Inteligencia Artificial.

3. Los desarrolladores de juegos son tecnológicamente inteligentes y trabajan duro para mantenerse al día con la tecnología. El título de programador de Inteligencia Artificial es ya un componente más en los equipos de desarrollo de videojuegos.

4. La industria del videojuego genera y mueve grandes cantidades de dinero. Actualmente se gasta incluso más que en las películas de cine.

5. El hardware de los juegos se va a proporcionar barato, con una potencia de cálculo de gama alta para la Inteligencia Artificial en los juegos de los próximos cinco años. Con los nuevos PC's y las nuevas generaciones de consolas que son capaces de mover todos los gráficos sin la CPU, no sería de extrañar que en unos años se dediquen a implementar procesadores de Inteligencia Artificial en

los videojuegos, lo que permitirá poder desarrollar muchas más técnicas ya que un procesador independiente puede mover todas las órdenes y hechos.

6. Los videojuegos necesitan ayuda del estudio de la Inteligencia Artificial. En la actualidad la importancia de la Inteligencia Artificial se basa en el comportamiento humano para unas situaciones muy limitadas, y las técnicas usadas (como funciones en C, o máquinas de estado finitas) no podrán mejorarse en un futuro. Por tanto, al igual que los videojuegos han ido mejorando gráficamente hasta llegar a unos puntos muy reales del mundo físico, se espera que se desarrollen personajes humanos más reales, utilizando nuevas técnicas y sistemas.

Además de estos motivos, lo que sigue moviendo a implementar técnicas de Inteligencia Artificial y mejorarlas es que dichas técnicas todavía no son perfectas. Con el crecimiento del sector de los videojuegos, se empieza a notar en el campo de la Inteligencia Artificial que se necesitan cada vez más expertos, donde las nuevas arquitecturas hardware (los chips con varios núcleos de proceso) abren nuevas puertas a la Inteligencia Artificial, al permitir tener varias unidades de proceso volcadas en el estudio del comportamiento inteligente.
No es muy descabellado pensar que en un cierto tiempo, el número de expertos en Inteligencia Artificial en el proyecto de desarrollo de un videojuego llegue a ser el mismo que el número de desarrolladores del videojuego en sí.

Después de esta breve introducción, y tras conocer los motivos que han llevado a la Inteligencia Artificial al mundo de los videojuegos, haremos una clasificación de géneros de videojuegos y haremos un repaso de las técnicas utilizadas en Inteligencia Artificial y que posteriormente se aplicarán en los videojuegos.

2.4 Clasificación de videojuegos

En este apartado, y basándonos en la clasificación de los videojuegos por la relevancia de su Inteligencia Artificial [Laird and van Lent, 2000], comentaremos los diferentes géneros existentes en los videojuegos que más tienen que ver con Inteligencia Artificial.

2.4.1 Videojuegos de acción

En los videojuegos de acción controlamos a un personaje en un entorno virtual en el que se mueve y dispara a los demás enemigos para combatir el mal. Estos juegos tienen dos perspectivas generalmente: una visión en primera persona donde el usuario ve como si fuera el jugador; o en tercera persona, donde se ve al personaje detrás de su espalda o sobre el hombro. Hay que destacar un subgénero dentro de este, donde se les llama a los videojuegos como *FPS* (**First Person Shooter**). Estos videojuegos, son denominados de *disparos en primera persona*.

En este tipo de videojuegos, la Inteligencia Artificial está presente en los enemigos, utilizando técnicas para recrear comportamiento humano en ellos. A pesar de que el punto de competencia entre los videojuegos de acción se basa en el realismo de los gráficos, la Inteligencia Artificial empieza a tener bastante importancia para ser otro punto de comparación y calidad en estos videojuegos. Algunos de estos videojuegos son *Half-Life, Tomb-Raider, Descent, Crysis* o *F.E.A.R.*.

2.4.2 Videojuegos de Rol

En este género, conocido también por las siglas *RPG* (**Role Playing Game**), el usuario puede ponerse en la piel de un guerrero, mago o ladrón entre otros. El jugador realiza misiones, recoge tesoros, pelea con monstruos y aumenta sus habilidades (rapidez, fuerza, magia, etc.) mejorando su experiencia. Algunos ejemplos de videojuegos más famosos son *Diablo, Baldur's Gate* o *Ultima.*

Como se mencionó anteriormente, el uso de Internet en los videojuegos ha generado un sub-género llamado *Rol Multijugador Masivo*, donde los usuarios juegan en el mismo mundo interactuando entre ellos. Algún juego de este sub-género es *Ultima Online* o *Everquest*.

Al igual que en los videojuegos de acción, las técnicas de Inteligencia Artificial se utilizan para controlar a los enemigos, compañeros de viaje, o personajes de apoyo (comerciantes o habitantes de ciudades).

2.4.3 Videojuegos de Aventuras

En estos juegos, los jugadores deben resolver puzzles e interactuar con otros personajes, donde se irá desarrollando la aventura que está determinada en parte por sus acciones. Los primeros videojuegos de este género estaban basados solamente en texto, como *Aventura* o *Zork*; pero los videojuegos actuales soportan gráficos en 3D (a veces utilizando motores desarrollados para algún videojuego de acción). Ejemplos de videojuegos con motores gráficos son *Monkey Island, Grim Fandango, King's Quest*.

En este género la Inteligencia Artificial suele ser usada para crear personajes de apoyo realistas, con los que el jugador debe de interactuar para progresar adecuadamente en el juego. La mayoría de estos videojuegos tienen unos scripts fijados y utilizan varios trucos para llevar al jugador hacia una historia lineal; pero sin embargo, en otros juegos, como *Blade Runner*, se ha incluido algo de autonomía y de scripts dinámicos en sus personajes y línea de la historia [Castle 1998], dando como resultado un argumento no lineal y teniendo, por ejemplo, distintos finales.

2.4.3 Videojuegos de Estrategia

En los videojuegos de estrategia, el usuario controla a un grupo de unidades (soldados, tanques, robots, aliens, etc.), para pelear contra ejércitos contrarios

controlados por el enemigo. Estas batallas suelen ser de varios tipos: históricas (*Close Combat*), de ficción (*Starcraft*), místicas (*Warcraft*), o de realidad alternativa (*Comand and Conquer*). El jugador a menudo se enfrenta a problemas de asignación de recursos, programación de la producción, o a la organización de las defensas y ataques [Davis 1999].

Las técnicas de Inteligencia Artificial en este caso se utilizan para dos funciones: la primera es para controlar a las unidades individuales que controla el usuario; y para controlar a un adversario estratégico que debe de desempeñar el mismo juego contra el usuario.

2.4.4 God Games (juegos de dios)

Los god games o simuladores de dios, ponen al jugador en el rol de una entidad que tiene el control total de un mundo virtual. El entretenimiento viene mediante la observación de los efectos de las acciones del jugador en los personajes, la sociedad y el mundo. *SimCity* es uno de los referentes de este género, donde el jugador humano actúa como alcalde y las técnicas de Inteligencia Artificial sirven para controlar a los ciudadanos de la simulada ciudad.

2.4.5 Simuladores sociales

El género recién expuesto presenta muchas similitudes con otro género que cuenta con un corto tiempo de vida, pero cuya aparición resultó revolucionaria, creándose desde entonces más videojuegos de este tipo. Estamos hablando de simulación social, género relacionado con este Proyecto Fin de Carrera. En muchos artículos de análisis de videojuegos se mezclan estos dos géneros, ya que la funcionalidad de ambos parte de un punto común (manejar todo al antojo del jugador). En este apartado mencionamos el videojuego *The Sims*, que se podría englobar dentro de los juegos de simulador de dios o simulador social, ya que se crea a un personaje, y se controla totalmente persiguiendo una serie de objetivos personales, como necesidades básicas o sociales. Disponen de autonomía propia si no lo controla el jugador durante un tiempo. El criterio que diferencia estos videojuegos como simuladores sociales radica en que el núcleo del juego lo forma el conjunto de las relaciones personales del personaje con otros personajes del juego.

2.4.6 Videojuegos deportivos

Dentro de este género podemos diferenciar claramente dos grupos de géneros: los juegos de deportes de equipo, y los juegos de deporte individual.

2.4.6.1 Videojuegos de deportes de equipo

Este tipo de videojuegos tienen al jugador humano como una combinación de entrenador y jugador de deportes populares como el fútbol, fútbol americano,

baloncesto, béisbol o hockey. El propósito de este tipo de videojuegos es conocido, ya que consisten en una representación virtual de los deportes en la realidad, por lo que las reglas de cada juego son conocidas.

La Inteligencia Artificial en los videojuegos de deportes de equipo está presente en dos tipos de funcionalidades que son similares a las de los juegos de estrategia comentados anteriormente. Una de ellas es el control todos los compañeros del equipo del jugador humano. Por lo general, el jugador humano controla a un jugador clave (como el quarterback en un equipo de rugby), o alterna entre los distintos jugadores del equipo, mientras la máquina controla todos los demás miembros del equipo. Una segunda función es la del oponente estratégico, como en los juegos de estrategia, pero siendo en este caso el entrenador oponente. Un aspecto llamativo en los videojuegos de este género es la inclusión de comentaristas, que le da color al juego [Frank 1999].

2.4.6.2 Videojuegos de deportes individuales

Para deportes de competición individual, como la conducción, snowboard, esquí, etc., el ordenador proporciona una simulación del deporte en cuestión con una perspectiva en primera o tercera persona. El jugador humano controla a un participante que compite contra otros jugadores humanos (en versión multijugador), o contra oponentes controlados por la máquina. Estos últimos oponentes se asemejan más a los de los videojuegos de acción, ya que la acción (en este caso la competición), se sucede en tiempo real. También pueden tener comentaristas, aunque se ven menos que en los juegos de deportes de equipo.

Como conclusión de este apartado del documento, diremos que, a pesar de enumerar estos géneros, existen otros muchos, y ciertos videojuegos pueden ser fusiones de varios géneros, como por ejemplo *Dungeon Keeper*, que compagina momentos de estrategia con momentos de acción.

Después de esta clasificación, nos centraremos en las técnicas de Inteligencia Artificial que se utilizan en los videojuegos, y algunas de las diferentes funciones que aplican estas técnicas y que tienen que ver con los géneros de videojuegos recién comentados.

2.5 Técnicas de Inteligencia Artificial

Una de las principales capacidades de la inteligencia humana es su capacidad para resolver problemas. La habilidad para analizar los elementos esenciales de cada problema, abstrayéndolos, el identificar las acciones que son necesarias para resolverlos y el determinar cuál es la estrategia más acertada para atacarlos, son rasgos fundamentales que debe tener cualquier entidad inteligente. Es por eso por lo que la resolución de problemas es uno de los temas básicos en Inteligencia Artificial.

En este apartado enfocaremos el repaso en las técnicas que se han aplicado en videojuegos, de los cuales se mostrarán ejemplos siguiente apartado. El objetivo de este apartado es el de dar a conocer y explicar las diferentes técnicas utilizadas en Inteligencia Artificial, para comprender posteriormente dichas técnicas a la hora de mencionar aplicaciones en videojuegos.

2.5.1 Algoritmos de búsqueda

Una de las técnicas más conocidas en Inteligencia Artificial son los algoritmos de búsqueda. Se trata de un proceso general de la resolución de problemas, donde la adquisición o recuperación de información, constituye parte de los mismos. En general un algoritmo de búsqueda se puede representar como el recorrido por un árbol en el que cada nodo representa un estado y cada rama representa las relaciones entre los estados cuyos nodos conecta. También se puede representar a su vez el árbol en forma de grafo.

Un problema se puede definir formalmente por cuatro componentes [Norvig and Russell, 2003]:

- Un **estado inicial**.
- Una descripción de las posibles **acciones** disponibles. La formulación más común usa la "función sucesor". Esta función, para un estado dado (llamado *x* por ejemplo), devuelve el conjunto de acciones del sucesor. Cada acción es una de las posibles acciones que se pueden aplicar en el estado *x*, y cada sucesor es el siguiente estado al que se puede llegar aplicando la acción al estado *x*. Con estos dos puntos ya podemos reflejar implícitamente el espacio de estados**.**
- **Prueba de meta**, que determina cuál estado es un posible estado final. A veces hay un conjunto de posibles metas, y la prueba simplemente verifica que el estado pertenece a ese conjunto.
- Un **coste de la ruta**, donde a través de una función devuelve un valor que será este coste. Esta función suele reflejar el desempeño de la acción. Por ejemplo, en un problema de caminos y ciudades, el coste puede ser los kilómetros que hay de un lugar a otro.

Con estos elementos ya tenemos definido un problema, que pueden ser reunidos en una estructura de datos para servir de entrada a un algoritmo de solución de problemas. Una solución al problema es la ruta del punto inicial al estado final. La calidad de la solución está medida por la función de coste de la ruta, donde una óptima solución contiene el valor más pequeño de ruta de todas las soluciones.

A continuación veremos distintos algoritmos usados, según sus características. Se agruparán en tres tipos generales: Búsqueda No Informada, Búsqueda Heurística (o Informada) y Búsqueda con Varios Agentes.

2.5.1.1 Búsqueda No Informada

Dentro de este grupo analizaremos los siguientes algoritmos: búsqueda en amplitud; búsqueda en profundidad y búsqueda bidireccional [Borrajo, Martínez, Juristo y Pazos, 1993].

- Búsqueda en profundidad: este algoritmo da prioridad a los nodos de niveles más profundos en el grafo de búsqueda. Su funcionamiento consiste en ir expandiendo todos y cada uno de los nodos que va localizando, por un camino concreto y de manera recurrente. Cuando ya no existen más nodos que visitar en dicho camino, regresa, de modo que repite el mismo proceso con cada uno de los hermanos del nodo ya procesado.

  Este tipo de algoritmo es efectivo cuando el número de soluciones es grande, están lejos del estado inicial y se tienen señales previsoras para indicar que una dirección candidata es errónea. Para ello se suele implementar una regla de parada que cuando se cumple, devuelve la atención del procedimiento a la alternativa más profunda que no exceda dicho límite. Para una mejor comprensión del algoritmo incluimos la siguiente figura ilustrativa (3):

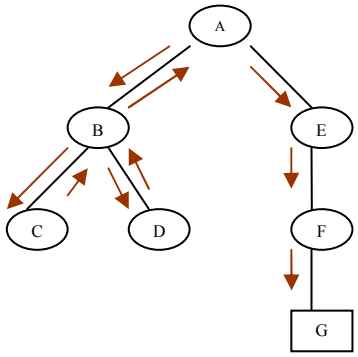

Figura 3. Desarrollo de nodos en una búsqueda en profundidad.

- Búsqueda en amplitud: A diferencia del algoritmo anterior, la búsqueda en amplitud o anchura comienza explorando los nodos adyacentes del nodo raíz. Por cada nodo adyacente, se exploran a su vez sus nodos adyacentes. Podría decirse que en este caso, el árbol o grafo es recorrido por niveles.

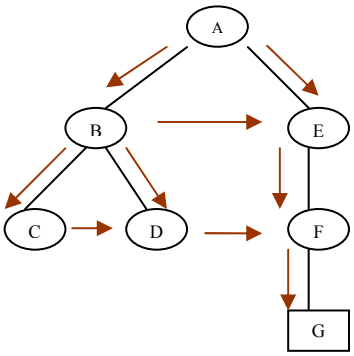

Figura 4. Desarrollo de nodos en una búsqueda en amplitud.

Fijándonos en esta figura, en primer lugar se exploran los nodos adyacentes del nodo raíz A. Después, se exploran los nodos de B y los nodos de E. Tras este paso, los nodos resultantes son C, D y F. Exploramos estos nodos, donde sólo F tiene un nodo adyacente a su vez, siendo el siguiente en ser explorado.

Este algoritmo es aconsejable cuando las metas están cercanas y son pocas. Expandir los nodos de forma uniforme garantiza encontrar la mejor solución de un problema de costo uniforme antes que ninguna, pudiendo devolver la primera solución ya que será la óptima, o bien expandir el resto de nodos del nivel de la solución encontrada para encontrar todas las soluciones posibles. Como inconveniente presenta el alto orden de complejidad computacional, que hace que, de no mantenerse muy limitados los nodos y niveles del grafo del problema, crezcan rápidamente los requerimientos y se vuelvan inaceptables.

- Búsqueda bidireccional: La idea de realizar una búsqueda bidireccional es la de realizar dos búsquedas simultáneas: una partiendo desde el estado inicial, y la otra desde el estado final, finalizando la búsqueda en el punto en que se encuentren. El único problema que presenta, es la de realizar una buena función, que permita calcular los predecesores para la búsqueda que empieza por el estado final. El caso más fácil de implementar resultaría cuando las acciones del espacio de estados fueran reversibles, esto daría que el predecesor y sucesor de un estado coincide.

- Hay otros tipos de búsquedas, que son de costes no uniformes. A continuación pondremos dos ejemplos de ellos:

  - El algoritmo de Dijkstra: este algoritmo explora los caminos más cortos que parten desde el nodo origen de un grafo y que llevan a todos los demás. Cuando se obtiene el camino más corto desde el nodo origen, al resto de nodos que componen el grafo, el algoritmo se detiene.

    El funcionamiento del algoritmo es el siguiente:

    - Teniendo un grafo dirigido ponderado de N nodos, con un nodo inicial x y un vector D de tamaño N que almacena las distancias desde x al resto de nodos.
    - Se inicializan todas las distancias de D con un valor infinito relativo ya que al principio son desconocidas.
    - Tomamos como nodo actual (a) el nodo inicial, y recorremos sus nodos adyacentes, excepto los nodos ya marcados (v_i).
    - Si la distancia desde x hasta v_i guardada en D es mayor que la distancia desde x hasta a sumada a la distancia desde a hasta v_i; esta se sustituye con la segunda nombrada, esto es: si ($D_i > D_a + d(a, v_i)$) entonces $D_i = D_a + d(a, v_i)$

- Se marca el nodo *a* como completo, y se toma como próximo nodo el de menor valor en *D*, volviendo al tercer punto, recorriendo los nodos adyacentes al nuevo nodo actual.

➢ Branch & Bound ("B&B", o "Ramificación y Poda"): Es un algoritmo que normalmente se realiza en profundidad, cuya característica se corresponde a que el algoritmo es capaz de detectar en qué ramificación las soluciones dadas ya no son óptimas. Esto se consigue cuando encuentra una solución, donde su coste se convierte en un límite (superior o inferior) para los siguientes nodos. De tal forma que si un nodo posterior tiene un coste desde el nodo raíz mayor/menor o igual que el límite, se termina la búsqueda por él.

2.5.1.2 Búsqueda Heurística

Una *heurística* consiste en una función matemática, definida en los nodos de un árbol de búsqueda, que sirve como estimación del coste del camino más corto de un nodo dado hasta el nodo final, y ayuda a resolver problemas incluso cuando no hay garantías de que nunca se va a ir por el camino equivocado. Cabe destacar que una heurística se descubre resolviendo problemas simplificados del problema real.

La heurística está reconocida como una de las aportaciones clave de la Inteligencia Artificial a la solución eficiente de problemas.

Como algoritmos principales de búsqueda heurística, citaremos los siguientes: Búsqueda en escalada, búsqueda en haz, algoritmo de mejor-primero, y A* [Borrajo, Martínez, Juristo y Pazos, 1993].

- Búsqueda en escalada: Es un algoritmo voraz, que no mantiene un árbol de búsqueda, sino sólo la representación del estado actual y el valor de su función objetivo. Por tanto, el algoritmo no mira más allá de los vecinos inmediatos del estado actual.

  Dado un estado actual, escoge el vecino del cual se obtiene un mejor valor de la función objetivo; y el algoritmo finaliza cuando alcanza un "extremo" (que es un máximo o mínimo, según el planteamiento del problema).

  La búsqueda en escalada no asegura encontrar la solución, pudiendo paralizarse en un máximo o mínimo local, o una meseta; a pesar de ello la solución de estas situaciones es la de permitir retroceder un paso o varios. Es capaz de encontrar soluciones rápidamente si la función es monótona, pero sin garantizar la optimalidad de estas soluciones.

- Búsqueda en haz: El nombre de este algoritmo proviene del hecho de que el proceso de búsqueda de una solución óptima supone calcular las distancias acumuladas a lo largo de los múltiples posibles caminos que nacen en cada punto de decisión local. Todos esos caminos, que parten de cualquier punto de decisión, constituyen un haz de caminos.

La idea básica es podar o recortar esos caminos del haz, y para ello se utiliza una búsqueda en amplitud, donde en cada nivel se realiza una clasificación en orden creciente de coste heurístico de los estados sucesores generados en dicho nivel. Después se aplica un umbral en el cual sólo se almacena un número de estados, que se corresponderán con aquellos estados con mejor valor. Aquellos que no han conseguido entrar dentro del umbral serán podados de la búsqueda.

Los componentes necesarios para realizar este algoritmo son: el problema a resolver representado por un grafo; y reglas heurísticas específicas al problema a tratar (para realizar el corte o poda y eliminar los nodos menos favorables).

- Ventajas: potencialmente puede disminuir el cálculo y tiempo de búsqueda. Al tener un límite de memoria no consume tanta como otros. La ventaja principal recae en la efectividad de las reglas heurísticas para analizar la poda o corte.
- Inconvenientes: no asegura encontrar la solución.

- Mejor-primero: Este método extiende la mejor trayectoria parcial en cada punto, considerando todos los nodos abiertos hasta el momento. La diferencia con el algoritmo de escalada es que este analiza varias trayectorias a la vez, siguiendo siempre la mejor conocida hasta el momento.
  Generalmente encuentra trayectorias más cortas hacia el nodo meta o final.

- A*: Este algoritmo, llamado A-estrella, evalúa los nodos combinando el coste para alcanzar el nodo actual *g(n)*, y el coste desde ese nodo al nodo final *h(n)*: $f(n) = g(n) + h(n)$
  Donde *f(n)* es el coste mínimo estimado para la solución por *n*.
  La optimalidad del algoritmo se basa en su heurística, donde la solución será óptima si la heurística es admisible (nunca sobrestima el coste para alcanzar el objetivo). Otras de las propiedades son: que es completo; es decir, si tiene solución, la encontrará; y que también es eficiente.

- Existen variantes al algoritmo A*, de las cuales mencionaremos dos: el algoritmo SMA* (Simplified memory A*), y el IDA* (Iterative Deepening A*) [Korf, 1985].
  El algoritmo SMA*, hace un uso más inteligente del espacio de almacenamiento y usa toda la memoria que disponga, utilizándola de manera óptima. Por tanto el coste temporal de este algoritmo se corresponderá con la cantidad de memoria disponible.

  El método IDA* consiste en un algoritmo de profundización iterativa en el que hace uso de la información heurística de que se dispone para el problema, para determinar los nodos a expandir, donde cada iteración es una búsqueda de "primero en profundidad". La profundización terminará

cuando se haya llegado a un nodo donde el coste de la función de evaluación sea mayor que el actual coste límite.

2.5.2 Búsqueda con Varios Agentes

Al tratar cierto tipo de problemas, en este caso juegos de dos jugadores, es necesario utilizar unos métodos de búsqueda distintos a los vistos anteriormente.

El desarrollo de este tipo de problemas se lleva a cabo por dos antagonistas que efectúan su acción cada uno mediante turnos. Estos problemas son denominados "de información completa", ya que un jugador conoce todos los datos de su rival en todo momento.

Por tanto, en este apartado veremos los algoritmos existentes para este tipo de juegos, donde mencionaremos el algoritmo MINIMAX, y la poda ALFA-BETA [Borrajo, Martínez, Juristo y Pazos, 1993].

Para definir formalmente un juego de dos jugadores, se puede componer de la siguiente manera:

- Un estado inicial que incluye el estado del tablero y qué jugador comienza.
- Un conjunto de operadores que indican las jugadas permitidas.
- Un estado de final de juego.
- Una función que asigna un valor numérico al resultado obtenido en el juego.

Con el problema definido podemos explicar los dos siguientes algoritmos.

- Algoritmo MINIMAX: La idea de este algoritmo es la de maximizar las jugadas del jugador considerando que el oponente minimizará todas. Esto se consigue a través de una función en el algoritmo donde se analizan todas las situaciones y es capaz de dar un valor numérico, donde las de número positivo serán para el maximizante, y las de número negativo para el minimizante. Esta función de evaluación se llama "estática". Se desarrolla una búsqueda por niveles hasta que se generan los nodos de un cierto nivel, y se aplica esta función de evaluación estática; posteriormente el valor devuelto viene dado por tres posibles situaciones:
  - La jugada no es ganadora para ningún jugador.
  - La jugada sale ganadora para el jugador maximizante (tendrá un número entero positivo).
  - La jugada sale ganadora para el jugador minimizante (tendrá un número entero negativo).

  Por ejemplo, en el caso del *Ajedrez* los posibles valores son (+1, 0, -1), que se corresponden respectivamente con ganar, empatar y perder. En el caso del *Backgammon* los posibles valores comprenden un rango de [+192, -192], correspondiéndose con el valor de las fichas.

Esta función presenta un problema a la hora de retornar el valor, y es que este valor no dice cómo fue determinado. Aún así, es posible mejorar este algoritmo usando la técnica que viene a continuación.

- Poda Alfa-beta: este algoritmo es un sencillo procedimiento basado en el algoritmo "B&B" mencionado anteriormente y que produce los mismos valores que el Minimax. Además mejora el Minimax de manera que ignora acciones que son incapaces de mejorar otras ya conocidas.
  Este algoritmo presenta tres características que lo hacen imprescindible en Inteligencia Artificial. Primero, complementa y suple el método Minimax, por lo mencionado en el párrafo anterior; incluso "Alfa-beta" indica a Minimax dónde debe detener la exploración de sucesores.
  En segundo lugar, este algoritmo ejecuta la evaluación de las hojas del árbol generado, calculando simultáneamente los valores de los nodos, con la generación del árbol. Esto es distinto al resto de algoritmos de búsqueda, ya que primero generan el árbol y luego lo evalúan.
  Por último, es un método muy comprensible y se demuestra correcto en lenguaje algorítmico.

  La poda Alfa-beta se basa en la idea de disponer dos valores que conformen una ventana a la cual deben pertenecer los valores de la función de evaluación para que sean considerados. En los nodos MAX, se debe maximizar el valor de los nodos sucesores (según el método minimax). En estos nodos, se utiliza el parámetro α que determina el máximo valor de los nodos sucesores. También, para los nodos MIN se utiliza otro valor, en este caso β, que almacena el valor mínimo de los valores de los nodos.

  La poda en los nodos MAX, se realiza si se cumple que el α del nodo MAX en una profundidad p, es mayor o igual que la β del nodo MIN de la profundidad $p-1$. $\alpha_p \geq \beta_{p-1}$

  La poda en los nodos MIN, se realiza si se cumple que la β del nodo MIN en una profundidad p, es menor o igual que el α del nodo MIN de la profundidad $p-1$. $\beta_p \leq \alpha_{p-1}$

2.5.3 Máquinas de Estado Finitas

Las Máquinas de Estado Finitas (FSM, **finite state machine**), también conocidas como Autómatas de Estados Finitos (FSA, **finite state automata**), son modelos de comportamiento de un sistema o un objeto complejo, con un número limitado de modos o condiciones predefinidos, donde existen transiciones de modo. Las máquinas de estado finitas son una técnica adoptada por la inteligencia artificial que se originó en el campo de las matemáticas, inicialmente utilizada para la representación de lenguajes.

Las FSMs están compuestas por cuatro elementos principales:

- Estados que definen el comportamiento y pueden producir acciones.
- Transiciones de estado que son movimientos de un estado a otro.
- Reglas o condiciones que deben cumplirse para permitir un cambio de estado.
- Eventos de entrada que son externos o generados internamente, que permiten el lanzamiento de las reglas y permiten las transiciones.

Una máquina de estado finita debe tener un estado inicial que actúa de punto de comienzo, y un estado actual que recuerda el producto de la anterior transición de estado. Los eventos recibidos como entrada actúan como disparadores, que causan una evaluación de las reglas que gobiernan las transiciones del estado actual a otro estado. La mejor manera de visualizar una FSM es pensar en ella como un diagrama de flujo o un grafo dirigido de estado.

Existen numerosas extensiones a las FSM y trucos para "mezclar" la secuencia y dificultar la predicción de sus actos. Una de estas opciones no deterministas implica la aplicación de otra técnica probada en la inteligencia artificial: Lógica Difusa, llamada Máquina de Estados Difusa (FuSM) [Martin and Thro, 1994].

2.5.4 Sistemas basados en reglas

La representación de un problema basado en reglas es muy importante en la Inteligencia Artificial, ya que por ejemplo estas reglas pueden ir implícitas en los cálculos del sucesor en los algoritmos de búsqueda, y tienen un funcionamiento muy parecido al razonamiento humano. Las reglas son una forma natural de representar el conocimiento, mediante la siguiente manera:

SI Antecedentes ENTONCES Conclusiones y/o Acciones.

Para representar estos problemas mediante reglas, surgen los Sistemas basados en reglas. Un Sistema basado en reglas está formado por tres componentes principales:

- Una base de hechos, que recoge el conocimiento sobre el dominio en un momento determinado.
- Una base de reglas de producción (denominado también base de reglas). Contiene la definición de la dinámica del dominio convenientemente formalizado y estructurado. La representación de esta dinámica se realiza utilizando reglas que relacionan una serie de premisas o antecedentes con sus consecuentes.
- Una máquina deductiva, llamada Motor de Inferencia. La inferencia consiste en una operación mediante la cual obtiene nuevo conocimiento a partir del existente.

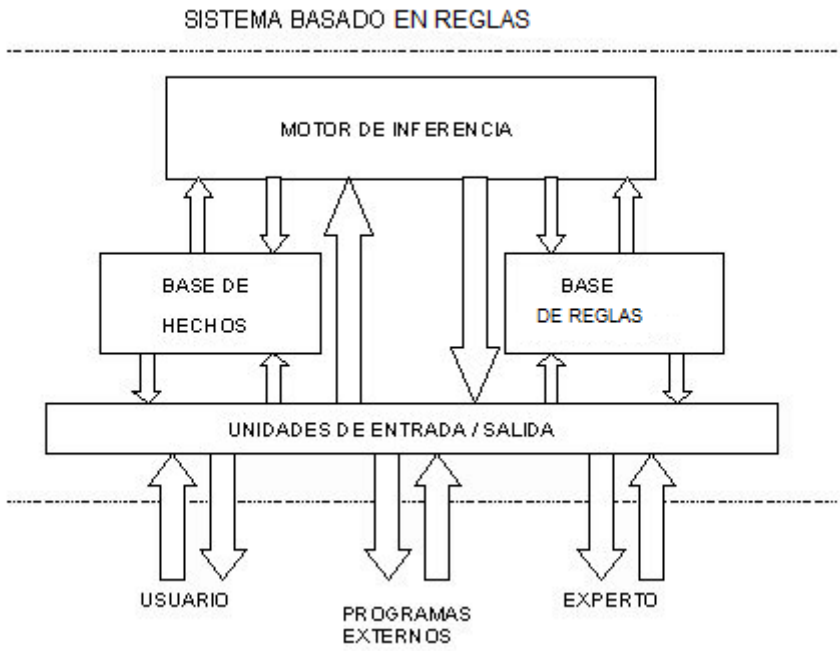

Figura 5. Sistema basado en reglas

Existen dos mecanismos fundamentales de encadenamiento de reglas, utilizadas para alcanzar el razonamiento o las conclusiones:

- Encadenamiento hacia delante: Se produce cuando partiendo del conjunto de hechos iniciales, se ejecuta una regla, generando nuevos hechos que permiten seguir disparando reglas sucesivamente, hasta llegar a una situación de parada (se alcanza la meta o no hay reglas activadas). Como interpretación del encadenamiento hacia delante diremos que:

  A, B, C ---> K

  "Si la base de hechos satisface A, B y C, entonces se añade el hecho K".

- Encadenamiento hacia atrás: Partiendo de un conjunto de metas, se buscan reglas que permitan concluir la meta, convirtiendo sus antecedentes en nuevas submetas. El proceso se repite hasta encadenar con la regla cuya conclusión satisfaga la meta inicial, o no existan reglas activadas. Como interpretación del encadenamiento hacia atrás diremos que:

  A, B, C ---> K

  "Si queremos obtener K, se debe obtener primero A, B y C".

  Por ejemplo, el lenguaje de programación PROLOG, sólo soporta el encadenamiento hacia atrás.

Los mecanismos de encadenamiento constituyen una parte importante del Motor de Inferencia. El Motor de Inferencia realiza siempre tres pasos:

1. Primero se realiza una fase de filtrado o equiparación de reglas: se seleccionan todas las reglas activadas, que son aquellas cuyo antecedente es satisfecho mediante la base de hechos.
2. Después aplica la reducción de conflictos: se selecciona mediante criterios adicionales (llamados estrategias de resolución de conflictos) un subconjunto de reglas del conjunto anterior. Lo habitual es aplicar secuencialmente varias estrategias hasta dar con una única regla.
   Algunas de estas estrategias son:
   - Refracción: Cada regla sólo se puede disparar una vez con los mismos elementos de la base de hechos.
   - Reciencia: se selecciona la regla que se satisface con los hechos más recientemente añadidos a la base de hechos.
   - Especificidad: seleccionar la regla que contenga más premisas.
   - Prioridad: seleccionar la regla con mayor prioridad.
   - Orden: se elige la primera regla según el orden en la base de reglas.
3. Se ejecuta la regla y se añaden los hechos y reglas derivadas a sus bases correspondientes.

Para el control de la activación de las reglas, se pueden cambiar las prioridades, o añadir premisas en las reglas para controlarlas.

2.5.5 Planificadores

La **planificación automática** es una disciplina de la Inteligencia Artificial que tiene por objeto la producción de planes (es decir, un planificación), típicamente para la ejecución de un robot u otro agente, el cual busca un plan para lograr cierto objetivo. El plan es una series de acciones a tomar para, partiendo de un estado inicial, cumplir el objetivo. Lo dicho suena muy parecido a la resolución de problemas, pero se considera a la planificación como diferente, ya que se utiliza una manera distinta de representar acciones, objetivos y estados. Además, difieren en la forma de construir y representar el curso de acción (secuencia de acciones).

Para representar problemas de planificación se utiliza un lenguaje particular (más restringido que la lógica de primer orden) y un algoritmo que permite operar sobre dicho lenguaje denominado **planificador**. La utilización de un lenguaje más restringido que el cálculo de predicados se debe entre otras cosas a propósitos de eficiencia. Un planificador típico considera 3 entradas:

- una descripción del estado inicial, que se representa por un conjunto de hechos que son verdaderos en un estado del mundo, escritos en un lenguaje adecuado.
- una descripción del objetivo u objetivos a alcanzar.
- un conjunto de acciones posibles, donde el planificador puede agregar acciones en cualquier momento, sin restringirse a una secuencia lineal.

Generalmente, cada acción especifica precondiciones que se deben cumplir como requisito a tal acción, así como postcondiciones, que constituyen el efecto sobre el estado actual del mundo.

Ejemplos de problemas de planificación pueden ser determinar la trayectoria de un robot en un espacion con obstáculos, o el problema de las torres de hanoi.

2.5.6 Aprendizaje Automático

El Aprendizaje Automático es una rama de la Inteligencia Artificial que trata de construir sistemas informáticos que implementen varias formas de aprendizaje, y que optimicen un criterio de rendimiento utilizando datos o experiencia previa.

Una situación en la que se requiere aprender es cuando no existe experiencia humana o cuando no es fácilmente explicable. Otra es cuando el problema a resolver cambia en el tiempo o depende del entorno particular. El Aprendizaje Automático transforma los datos en conocimiento y proporciona sistemas de propósito general que se adaptan a las circunstancias.

Entre las muchas aplicaciones conocidas pueden citarse el reconocimiento de voz o de texto manuscrito, navegación autónoma de robots, recuperación de información documental, sistemas de diagnóstico, etc.

Los diferentes algoritmos de Aprendizaje Automático se agrupan en función de la salida de los mismos. Algunos tipos de algoritmos son:

~ Aprendizaje supervisado

Este algoritmo deduce una función a partir de datos de entrenamiento. Los datos de entrenamiento consisten de pares de objetos (normalmente vectores): una componente del par son los datos de entrada y el otro, los resultados deseados. La salida de la función puede ser un valor numérico o una etiqueta de clase. El objetivo del aprendizaje supervisado es el de crear una función capaz de predecir el valor correspondiente a cualquier objeto de entrada válida después de haber visto una serie de ejemplos (los datos de entrenamiento). Para ello, tiene que generalizar a partir de los datos presentados a las situaciones no vistas previamente.

~ Aprendizaje no supervisado

Todo el proceso de modelado se lleva a cabo sobre un conjunto de ejemplos formado tan sólo por entradas al sistema. No se tiene información sobre las categorías de esos ejemplos.

~ Aprendizaje por refuerzo

El algoritmo aprende explorando el mundo que le rodea. Su información de entrada es el *feedback* o retroalimentación que obtiene del mundo exterior como respuesta a sus acciones.

~ Transducción

Similar al aprendizaje supervisado, pero no construye de forma explícita una función. Trata de predecir las categorías de los futuros ejemplos basándose en los ejemplos de entrada, sus respectivas categorías y los ejemplos nuevos al sistema.

~ Aprendizaje multi-tarea

Son métodos de aprendizaje que usan conocimiento previamente aprendido por el sistema, de cara a enfrentarse a problemas parecidos a los ya vistos.

2.5.7 Redes de Neuronas

Las Redes de Neuronas Artificiales (denominadas habitualmente como RNA), son un paradigma de aprendizaje y procesamiento automático inspirado en la forma en que funciona el sistema nervioso de los animales [Hilera y Martínez, 2000]. Se trata de un sistema de interconexión de neuronas en una red que colabora para producir un estímulo de salida.

Para ello se realiza una simulación de las propiedades observadas en los sistemas neuronales biológicos, a través de modelos matemáticos recreados por mecanismos artificiales como circuitos, un conjunto de válvulas, etc. El objetivo es que dicho estímulo de salida sea similar al que podría dar el cerebro.

La unidad de proceso de una RNA es la neurona. Existen tres tipos de unidades en cualquier sistema: entradas, salidas y ocultas. Las unidades de entrada reciben señales desde el entorno; las de salida envían la señal fuera de la red, y las unidades ocultas son aquellas cuyas entradas y salidas se encuentran dentro del sistema. Se conoce como capa o nivel a un conjunto de neuronas cuyas entradas provienen de la misma fuente y cuyas salidas se dirigen al mismo destino.

Las salidas emitidas por las neuronas vienen dadas por tres funciones:

- Función de Propagación (también conocida como función de excitación). Por lo general consiste en el sumatorio de cada entrada multiplicada por el peso de su interconexión (valor neto). Si el peso es positivo, la conexión se denomina excitatoria; si es negativo, se denomina inhibitoria. Una conexión excitadora quiere decir que si una neurona *i* está activada, la neurona *j* recibirá una señal que tenderá a activarse. Por otro lado, una conexión inhibitoria indica que si una neurona está activada, enviará una señal a la otra neurona de la conexión que hará que se desactive esta última.
- Función de Activación. Es una función que modifica la anterior y que también combina las entradas con el estado actual de la neurona para producir un nuevo estado de activación. Puede no existir, siendo en este caso la salida la misma función de propagación.

- Función de Salida o de Transferencia. Se aplica al valor devuelto por la función de activación. Se utiliza para acotar la salida de la neurona y generalmente viene dada por la interpretación que se quiera proporcionar a dichas salidas. Los estados del sistema en un tiempo *t* se representan por un vector *A(t)*. Los valores de activación pueden ser continuos o discretos, limitados o ilimitados. Si son discretos, suelen tomar un conjunto discreto de valores binarios, así un estado activo se indicaría con un 1 y un estado pasivo se representaría por un cero. En otros modelos se considera un conjunto de estados de activación, en cuyo valor entre [0,1], o en el intervalo [-1,1], siendo una función sigmoidal.

A continuación enumeraremos algunas de las ventajas que proporciona este paradigma.

Tolerancia a Fallos: Comparados con los sistemas computacionales tradicionales (los cuales pierden su funcionalidad en cuanto sufren un pequeño error de memoria), en las RNA si se produce un fallo en un pequeño número de neuronas, aunque el comportamiento del sistema se ve influenciado, no sufre una caída repentina.

Operación en Tiempo Real: Los computadores neuronales pueden ser realizados en paralelo, y se diseñan y fabrican máquinas con hardware especial para obtener esta capacidad.

Otra de las ventajas proporcionadas por las RNA y una de las características más atractivas, es la del Aprendizaje Adaptativo: Es la capacidad de aprender a realizar tareas basadas en un entrenamiento o una experiencia inicial.

Auto organización: Las RNA usan su capacidad de Aprendizaje Adaptativo para organizar la información que reciben durante el aprendizaje y/o la operación. Una RNA puede crear su propia organización o representación de la información que recibe mediante una etapa de aprendizaje. Esta auto organización provoca la facultad de las RNA de responder apropiadamente cuando se les presentan datos o situaciones a los que no habían sido expuestas anteriormente.

2.6 Aplicaciones de las técnicas de Inteligencia Artificial

En este apartado analizaremos diversas aplicaciones, como los Non Player Characters y *bots*, la búsqueda de caminos (Pathfinding), o las aplicaciones de Aprendizaje Automático entre otros, viendo algún ejemplo de videojuego donde se usan y comentando la técnica implementada para cada caso.

2.6.1 Non Player Characters y *bots*

La Inteligencia Artificial ayuda a los videojuegos a modelar el comportamiento de los NPC (Non Player Character) [Mac Namee, Cunningham, 2001], o personajes de apoyo. Estos NPC son aquellos personajes en un videojuego que son controlados por

Inteligencia Artificial u otra técnica. Donde más común es ver este tipo de personajes es en los RPG's (juegos de rol), o en los videojuegos de aventuras.

Uno de los factores relevantes en este tipo de juegos es el de las batallas contra los enemigos, ambientadas contra monstruos fantásticos que habitan en los diferentes hábitats del mundo. Cuando el jugador está viajando a través del mundo, puede enfrentarse contra alguno (o algunos) de estos monstruos y si logra derrotarlos aumentará sus puntos de experiencia. En la actualidad en este tipo de peleas la inteligencia de los NPC se reproduce mediante scripts o máquinas de estado finitas. Utilizando una técnica de redes neuronales, es posible emular una mayor capacidad de inteligencia de ciertos aspectos de los NPC. Como resultado de esto obtenemos una mayor variedad para el juego y mayor entretenimiento para el jugador, ya que los NPC nunca atacarán de la misma manera y podrán ir aprendiendo las diferentes técnicas que el jugador utilice para su ataque.

En los videojuegos de aventuras, los NPC actualmente están implementados de manera que constan de una serie de respuestas que devuelven al usuario, según determinados menús o palabras clave. Uno de los videojuegos más complejos en este campo, es *Blade Runner* [Castle 1998]. Los personajes tienen cierta autonomía y algunas metas sencillas, pero que son muy estrechas con los objetivos del juego, donde los comportamientos para alcanzar estos objetivos son muy limitados.

Estos personajes deben existir en un mundo virtual; y en general, desempeñar un papel humano en ese mundo. Para ello deben interactuar con el ambiente que los rodea y con los demás personajes, tanto humanos como otros personajes controlados por el juego, dando respuestas humanas, o realizando una comprensión del lenguaje natural. Ya que este tipo de personajes son los que más deben parecerse a humanos, requieren una amplia gama de capacidades de Inteligencia Artificial.

Otra de las aplicaciones es la denominada *bot*. Un *bot*, en el ámbito de los videojuegos, es un programa que es capaz de jugar por sí mismo a un juego. La calidad del *bot* en este caso viene determinada por su capacidad de vencer (y en qué términos) el videojuego. Los géneros de videojuegos más conocidos donde se utilizan los *bots* son en los CRPG (*computer role-playing games*), ya que este tipo de aplicaciones requieren una gran capacidad de estrategia para ganarlos; y también en los FPS, en los que sustituyen a un jugador humano cuando no hay contrincantes disponibles o en juego offline.

Un ejemplo donde se utilizan *bots* es *The Sims*. En este juego, los *bots* actúan de acuerdo a su perfil psicológico y necesidades a cubrir en ese momento, lo que aporta una gran riqueza de posibilidades y acciones al videojuego.

Igualmente, las técnicas de Aprendizaje Automático y Redes de Neuronas aplicadas a los *bots*, permiten a éstos adaptarse a los cambios de las condiciones de su medio ambiente e interactuar por los escenarios con un conocimiento mínimo. Por ejemplo, en un videojuego donde hay una persecución de automóviles, el automóvil contrario (controlado por la máquina), no sabrá cómo es el escenario ni el perfil del jugador. Según avanza el juego, la máquina va creando un perfil del jugador y va aprendiendo la composición del escenario. Esto provocará un aumento de la dificultad,

ya que la máquina irá modificando su estrategia, lo que influirá en el jugador humano a tomar decisiones distintas, consiguiendo que el juego no le resulte monótono.

2.6.2 Pathfinding

Muchos de los videojuegos cuentan con mapas. En algunos juegos, los NPC y *bots* deben buscar al jugador humano para evitar que termine la pantalla o continúe su objetivo; y para ello se utilizan técnicas básicas y habituales como el *algoritmo A** o el *algoritmo de Dijkstra*.

En los primeros videojuegos, las tácticas de los enemigos eran simplemente la de moverse de un lado a otro del escenario, o perseguir constantemente al jugador hasta acabar con él; esto se controla con simples máquinas de estado finitas. En esos primeros juegos, los enemigos se hacían más competitivos, no por mejora de la inteligencia, si no por tener habilidades superiores o armas más potentes. Pero desde algunos videojuegos recientes, se utilizan, por ejemplo, algoritmos de búsqueda o planificadores, comentados anteriormente, que proporcionan a los enemigos un comportamiento más humano [Rabin, 2003].

Las Redes de Neuronas y los métodos de Aprendizaje Automático también ayudan a los problemas de *pathfinding*. En la actualidad, los problemas de encontrar caminos o lugares donde esconderse son realizados sólo mediante la codificación explícita previa de las reglas correspondientes. La aplicación de los agentes autónomos y las redes neuronales logrará crear NPC y *bots* que interactúen dinámicamente con el terreno en el que se encuentren. Esto, mediante el aprendizaje y la percepción del medio.

A continuación mostramos distintas estrategias o tácticas en videojuegos, que utilizan las técnicas de *pathfinding*:

2.6.2.1 Movimiento táctico

En videojuegos como *Halo* y *Killzone*, los caminos se calculan con una heurística que tiene en cuenta los factores del combate. Algunos factores pueden ser que el actor se mueva en el radio de fuego del jugador, o que esté expuesto a las armas del jugador.

Se utilizan también los algoritmos de camino más corto, que pueden ser modificados para tener en cuenta estos factores, pero pueden provocar una mayor lentitud de la búsqueda.

2.6.2.2 Selección de ubicaciones para cubrir al jugador

Muchos videojuegos de FPS utilizan sitios donde los jugadores se pueden cubrir. Son tanto calculados automáticamente como anotados por los desarrolladores. Durante la ejecución del juego, los actores seleccionan esos puntos teniendo en cuenta diversos factores como: la distancia entre el actor y el lugar de cubrimiento; si el

camino hasta ese punto implica encontrarse con un jugador que pueda dispararle; y la posición en relación con otros actores del mismo equipo, o con el jugador.

2.6.2.3 Análisis de cuello de botella

Muchos videojuegos utilizan una interfaz gráfica donde pueden observarse las distintas conexiones entre habitaciones o el mapa en general. Utilizando esa información, se puede identificar qué partes del escenario son cuellos de botella donde el jugador debe pasar. Por ejemplo *Half-life* utiliza esta técnica para ayudar a mejorar la colocación de los NPC y de sus movimientos durante los combates con el jugador.

2.6.3 Unidades

En los videojuegos de estrategia, *God Games* (simuladores de dios), o deportivos, las técnicas de Inteligencia Artificial se emplean para controlar las unidades individuales. Estas unidades se controlan mediante máquinas de estado finito o grandes funciones en C, que se incrementarán con rutinas especiales para planificar la ruta. Además de obedecer las órdenes, las unidades deben tener una cierta capacidad para actuar de manera autónoma. Por ejemplo, un pelotón de soldados, ante la decisión de pasar de una posición a otra, no debe de hacer caso omiso a una unidad enemiga que esté en el camino, sino que debe de tener la autonomía suficiente como para decidir si atacar o no. Este tipo de comportamientos requiere un razonamiento de sentido común.

Dado que en los últimos juegos hay cientos de unidades activas a la vez, la importancia computacional y de memoria, cobra importancia dentro de la Inteligencia Artificial para controlar a estas unidades al mismo tiempo.

2.6.4 Uso eficiente del trabajo en equipo

En los videojuegos de acción, hasta ahora los *bots* son creados y colocados en los diferentes escenarios para que el jugador se encuentre con ellos. El comportamiento de estos es totalmente autónomo con respecto al de sus compañeros. Con la aplicación de redes de neuronas y aprendizaje se podrán crear *bots* que interactúen entre sí para lograr un mismo fin, ya que esto es uno de los aspectos básicos de los agentes inteligentes.

2.6.5 Otras aplicaciones

- La Inteligencia Emergente: es un concepto que ha empezado a implementarse recientemente. Está presente en videojuegos cuyo protagonismo recae en unas criaturas, que van aprendiendo y actuando en consecuencia de las decisiones del jugador, y donde también crecen o evolucionan acorde a su aprendizaje. Estas técnicas

se inspiran en el funcionamiento de la naturaleza, por lo que da la sensación de que la evolución de las criaturas es real y no simulada.

- "Inteligencia Estafadora". Determinados juegos utilizan técnicas para detectar cuándo el jugador humano se encuentra en una situación ventajosa contra la máquina, y entonces se activan para aumentar la complejidad del juego y favorecer a la Inteligencia Artificial. Estas técnicas son habituales en videojuegos de deportes.

Como conclusión de este apartado, cabe decir que existen otras aplicaciones de la Inteligencia Artificial para la implementación de enemigos inteligentes. Debido a la extensa geografía del entorno, unas buenas técnicas para utilizar son, por ejemplo, planificadores de caminos, razonamiento espacial y temporal, etc. La evolución de los videojuegos provoca que vayan aumentando su complejidad, y los enemigos necesitan usar técnicas de aprendizaje, adaptarse a las tácticas y estrategias de sus enemigos, y esas respuestas deben de darse dentro de un tiempo de reacción igual de realista al de un humano.

2.7 Videojuegos que utilizan técnicas de Inteligencia Artificial

En este apartado comentaremos algunos de los videojuegos que se han mencionado a lo largo de este documento, para dar a conocer alguna de las aplicaciones que usa y técnicas utilizadas.

2.7.1 Quake II

Quake II es un videojuego de disparos en primera persona desarrollado por *id Software* y distribuido por *Activision* en 1997. El código fuente de *Quake II* fue posteriormente liberado por *id Software* bajo la licencia GPL. Esto provocó la realización de varios proyectos o investigaciones. Una investigación realizada [de Laird and van Lent, 1999; Laird, 2000] sobre *Quake II*, se basó en la construcción de enemigos con los mismos puntos débiles y fuertes que los jugadores humanos. Se implementó un sistema, llamado *Soar Quakebo*t, el cual es un sistema experto en tiempo real donde almacena múltiples objetivos, tácticas y conocimiento del juego, formado en total por más de 800 reglas. Mientras se explora el escenario, el sistema va creando un modelo interno de ese mundo y lo utiliza para plantear sus tácticas (localizar a un enemigo, hacer una emboscada), así como la recogida de objetos de armas y salud.

Existe además un proyecto llamado *Quakebot C/S*[1], donde hay implementado un núcleo y distintos desarrolladores pueden incluir sus módulos de Inteligencia Artificial. El núcleo central se encarga de las comunicaciones y de las funciones básicas de los Bots. La Inteligencia Artificial del bot consiste en conjuntos de reglas que varían de Bot a Bot. Estos conjuntos están modulados y son reemplazables, de manera que

[1] http://www.unconventional-wisdom.org/QuakeBot/quakebot.htm

cambiándolo, el comportamiento del bot puede ser totalmente distinto. Existe una parte especificada de la arquitectura, para que la gente individualmente pueda programar sus heurísticas, que son compatibles con la interfaz. En este módulo se pueden acceder a rutinas de niveles inferiores. Todo ello está implementado en C o C++.

Por último, existe otro proyecto creado en torno a *Quake*, llamado *Quake II Neuralbot*[2]. *Neuralbot* es un *bot* para el juego *Quake II* que utiliza una Red Neuronal para decidir su comportamiento y un algoritmo genético para el aprendizaje.

2.7.2 Halo

Halo es un videojuego de acción en primera persona, o *FPS*. En *Halo* (lanzado en Europa en 2002), los enemigos utilizarán todos los recursos a su alcance en cuando vean al jugador, pudiendo llamar a otras tropas de soldados, o incluso tanques para acabar con el jugador. También se pueden dar escenas en las que dos tropas distintas a la del jugador se estén enfrentando, y éste tenga que pasar inadvertido, o irán todos a por él.
Durante el diseño de *Halo*, el modelo inicial para el estado del actor se basaba en lógica difusa, con:

- Actores con cuatro tipos de estados de ánimo (miedo, rabia, actitud defensiva y sorpresa).
- Cada variable tiene sus umbrales.
- Existen determinados eventos que afectan a esas variables.
- Y las acciones serán impulsadas mediante esas variables.

Uno de los propósitos de *Halo* es el conseguir ser un juego imprevisible (que el juego no sea repetitivo, dando la impresión al jugador que cada vez que se juegue sea distinto). Para conseguir ser imprevisible, se apostó por utilizar técnicas de Inteligencia Artificial en las que los NPC no reaccionaran de la misma manera ante el jugador.

Uno de los defectos respecto a su Inteligencia Artificial es su escasa capacidad de ésta para hacer frente a objetivos con rapidez, por lo que se optó por proporcionar a los NPC poderosos ataques cuerpo a cuerpo para compensar dicha capacidad.

Existen cuatro tipos de información que un actor desea almacenar: pre-comportamiento persistente, pre-comportamiento de corto plazo, pre-objeto, y pre-objeto por comportamiento. Las diferencias entre las dos primeras se basan en que la información se almacena en variables, o es liberado de forma dinámica durante la ejecución de la aplicación. La diferencia entre los dos últimos tipos se basa en la localización de objetos del escenario, y los objetos que se deben de utilizar después de un enfrentamiento, por ejemplo.

Toda esta información se almacena en estructuras de datos. Y como explica Isla Damian[3]: “En *Halo*, estas representaciones se denominan objetos, y su principal

[2] Disponible para descarga en http://homepages.paradise.net.nz/nickamy/neuralbot/index.html

función es un repositorio de información sobre la percepción de objetos en el mundo. Tras esta información de estado (posición, orientación, ubicación, pathfinding, etc) distinto del estado del mundo real y con acceso controlado por el actor, la percepción permite que las dos representaciones en ocasiones divergen. Por lo tanto, el actor puede creer cosas que no son verdad, y entrar en el reino de la Inteligencia Artificial, que puede ser engañado, confundido, sorprendido, decepcionado, etc."

2.7.3 F.E.A.R.

Este videojuego, lanzado en el año 2005, se destapó como una gran innovación, quitándose el cartel de ser un mero clon de otros videojuegos *FPS*. Uno de los aspectos a destacar, es que el juego sólo ha contado con un programador de Inteligencia Artificial.

En el núcleo del juego, y centrándonos en su Inteligencia Artificial, *F.E.A.R.* utiliza, FSM (Máquinas de Estado Finitas) para controlar el comportamiento de un personaje, y algoritmos A* para establecer caminos o rutas. Además de estas dos técnicas, utiliza *Real-Time planning* (planificación en tiempo real), permitiendo la mejora del desarrollo de comportamientos de los personajes.

En su conjunto, la Inteligencia Artificial en *F.E.A.R.* permite que los NPC, entre otras acciones, sean capaces de cubrirse, disparar a ciegas, atravesar ventanales, eliminar grupos de oponentes lanzando granadas o comunicarse con compañeros de equipo.

Los agentes en *F.E.A.R.* [Orkin, Jeff] se componen de una pizarra (blackboard), una zona de memoria dinámica, varios subsistemas (navegación, animación, sistema de armas, etc.) y sensores. Los sensores son los encargados de detectar cambios en el entorno, almnacenando estas percepciones en la zona de memoria. El planificador usa estas percepciones para guiar la toma de decisiones y comunicar instrucciones a los subsistemas por medio de la pizarra.

Los sensores pueden percibir estímulos tanto externos (visuales, auditivos), como internos (dolor). Algunos están dirigidos por eventos mientras que otros sondean el entorno. Los primeros son útiles para reconocer al instante sonidos o daños, y los segundos para extraer información del mundo, por ejemplo, para ir generando una lista de posibles posiciones tácticas. Algunos de estos sensores son:

- SensorSeeEnemy: el sensor que se encarga de los estímulos visuales.
- SensorNodeCombat: observa el entorno tres veces por segundo, buscando sitios potenciales para esconderse o disparar desde posiciones protegidas. Establece una lista de posibles sitios y los ordena por su distancia con el personaje.
- PassTarget: encuentra una ruta hasta una posición táctica conocida, y determina si está libre de peligro.

[3] Isla Damian es un ingeniero del equipo de Halo.

La toma de decisiones se basa en un planificador en tiempo real. Cuando un sensor detecta cambios significantes en el estado del entorno, el agente evalúa de nuevo la relevancia de sus objetivos, puesto que sólo un objetivo puede estar activo en un instante determinado. Cuando el objetivo principal cambia, el agente usa el planificador para encontrar la secuencia de acciones que conduzcan a satisfacerlo.

Una acción se activa modificando algunos valores en las variables de la pizarra. Los subsistemas se actualizan periódicamente, y cambian su comportamiento de acuerdo a las instrucciones de la pizarra. Por ejemplo, la acción GotoTarget establece un nuevo destino en la pizarra, y el sistema de navegación, al actualizarse, debe encontrar una ruta hasta el nuevo destino.

2.7.4 Crysis

Este videojuego trata de otro FPS de última generación (lanzamiento en 2007), y se basa en las más avanzas tecnologías de la industria.
En particular *Crysis* pretendía hacer que los personajes fueran mucho más reales. Para ello hay que limitar un poco las animaciones, lo que puede hacer a los personajes menos creíbles, pero tampoco supone un problema grave, ya que se solucionó con un buen motor gráfico que permitió no interferir en el desarrollo de la Inteligencia Artificial.

El lema de diseño de *Crysis*, remontado a *FarCry*[4], es que todas las batallas deben de ser exclusivas, ninguna igual. La idea es usar grandes escenarios que inviten a la variación, pero sobre todo lo que juega el papel más importante, es que el comportamiento del jugador es la única fuente de incertidumbre, que hace el comportamiento del personaje menos monótono, aunque pueda ser previsible. A esto le llaman Inteligencia Artificial dinámica. Se consigue con la construcción de una arquitectura global de Inteligencia Artificial con un sistema sensorial, donde la lógica debe de ser personalizada. También permite dejar que el comportamiento salga de la interacción con el mundo.

2.7.5 Los Sims

Este primer juego (lanzado en el año 2000) de una consagrada saga supuso, y sigue siendo, uno de los juegos más influyentes destacados por la utilización de técnicas de Inteligencia Artificial.

Los objetos desempeñan un papel fundamental en el juego. No sólo aportan objetivos, sino que también permite crear un ambiente rico en el que los personajes interactúan con él.

[4] Juego anterior a Crysis de la misma compañía. http://es.wikipedia.org/wiki/Crysis

Debido a la cantidad de objetos que abundan en este videojuego, cada uno tiene su propio micro-hilo de ejecución. Esto permite que a la hora de insertar un nuevo objeto en el escenario se ejecute una función de inicialización del objeto. Además difiere de los demás juegos donde los objetos no tienen lógica propia (la mayoría lo incluyen dentro del motor del juego).

Los personajes en *Los Sims* son cada vez más sofisticados en la evolución de la saga, pero hay muy pocos detalles sobre el funcionamiento interno de la Inteligencia Artificial.

Una forma de lógica difusa se utiliza para modelar las emociones de los actores. En cada uno de ellos se han medido los sentimientos básicos en un rango [-100, 100] teniendo:

- Física: hambre, comodidad, higiene y vejiga.
- Emocional: energía, diversión, relación social y habitación.

Cada uno de ellos se asigna a un valor de estado o felicidad utilizando una curva de respuesta. Algunas funciones son lineales, por ejemplo, la higiene, la energía o la felicidad, pero otros son funciones polinómicas. Por ejemplo, los actores más hambrientos se preocupan menos de otras cosas, y más por el hambre.

Teniendo en cuenta estos sentimientos básicos, la toma de decisiones por parte de la Inteligencia Artificial se basa en encontrar los objetos que cumplan con sus necesidades actuales. Esto se hace sobre la base de varios factores:

- Las acciones posibles sobre la lista de los objetos que están disponibles.
- La recompensa que ofrece cada una de las acciones al actual actor.
- La distancia del objeto a la ubicación actual (problema de pathfinding).
- El nivel de necesidad que siente el actor en el momento.

El comportamiento de los actores de *Los Sims* es por tanto un subproducto de la toma de decisiones y los objetos inteligentes. De hecho, todos sus planes son fijos o predispuestos, lo que significa que se especifican manualmente utilizando scripts. Por ejemplo, el proceso de preparar el almuerzo puede requerir varios pasos, pero cada uno de ellos se especifica a través de un script adjunto a la nevera.

2.7.6 Black & White

Black & White propone la tarea de convertirte en un dios, donde deberás elegir un bando (bueno, malo o neutral). El objetivo es conquistar las aldeas que hay en la isla. Para ello contaremos con la ayuda de una criatura.

La criatura es un elemento muy importante del juego. Empezaremos con una criatura pequeña, a la que tendremos que amaestrar según nuestro criterio. Puede ser tanto buena, o mala, según las decisiones que hagamos, y lo que le digamos a la criatura que esté bien o mal. Posteriormente crecerá y se comportará en consecuencia de este aprendizaje, y sus acciones se verán magnificadas. En este aspecto, las técnicas utilizadas para el aprendizaje de la criatura se basan en redes neuronales y árboles de búsqueda, utilizadas con gran éxito.

2.7.7 Half-Life

Este videojuego es un ejemplo de los que la Inteligencia Artificial está implementada con un sistema basado en reglas. Los NPC ya cuentan con comportamientos prefijados, es decir, los personajes no utilizan aprendizaje automático para ir aprendiendo de los movimientos del jugador y establecer nuevas estrategias contra él; sino que los programadores tuvieron en cuenta cualquier decisión o comportamiento del jugador, para aplicar reglas a esos comportamientos y actuar en consecuencia.

2.7.8 Counter Strike

Counter-Strike es un videojuego de disparos en primera persona por equipos. Se concibió originalmente como un juego de tipo multijugador (ya sea en LAN u online). *Counter-Strike* es una modificación completa del juego *Half-Life*, cuya primera versión fue lanzada el 18 de junio de 1999. La última versión del juego es el *Counter-Strike: Online*, que desde su salida oficial el 15 de septiembre de 2008 ha cosechado un gran éxito en Internet, haciendo de él el juego de acción en primera persona online más jugado.

En su primera versión, el juego sólo disponía de la versión en red (online o por red local), lo que suponía no poder disfrutar del juego en modo offline. Esto suponía una desventaja ya que por aquellos años no era tan frecuente disfrutar de una conexión a Internet, y mucho menos de una conexión con un ancho de banda lo suficientemente aceptable. Debido a la gran acogida del juego pero considerando esta desventaja, los desarrolladores plantearon realizar una nueva versión con modo offline. Para ello se introdujo la tecnología multiagente, (conocida como *bots*), que manejaban los personajes del juego simulando la actuación de otros jugadores humanos.

Los primeros agentes eran muy sencillos ya que interactuaban de forma muy limitada con el entorno del juego. Actuaban en solitario por el mapa, guiados por puntos ya prefijados que indicaban por dónde podían moverse. Esto no se acercaba a conseguir un comportamiento humano en los *bots*. De hecho, en estas primeras versiones la precisión de sus disparos era tan perfecta (debido a que era más fácil

simular un disparo perfecto que la puntería humana), que el jugador era eliminado con un único disparo de un *bot*, desde distancias que lo hacían prácticamente invisible al ojo humano.

Con las posteriores versiones, los agentes fueron evolucionando, consiguiéndose metas como:

- Comunicación entre los miembros del equipo para definir estrategias o seguir órdenes dictadas por el jugador humano, pudiendo decidir si obedecer dichas órdenes o descartarlas.
- Analizar en tiempo real el terreno y a simular la vista humana dentro del juego, así como "empeorar" la puntería para conseguir unos agentes más realistas.

Las características en general de los agentes en el juego son las siguientes:

- Perciben el entorno: Los agentes reciben información del entorno, posición de objetos, procedencia de sonidos, etc.
- Interactúan con el entorno: Los agentes se mueven en un mundo virtual con unas condiciones físicas, estos pueden manejar objetos (como abrir puertas, romper ventanas) o atacar a otros agentes.
- Se comunican entre ellos: Los agentes pueden usar tanto un chat como comandos de radio que les permiten organizarse.
- Se dirigen por un conjunto de tendencias buscando satisfacer unos objetivos: Dependiendo del equipo al que pertenezcan y el tipo de mapa intentaran realizar unas determinadas acciones para ganar la partida.
- Poseen sus propios recursos: Los agentes, como cualquier otro jugador, al principio de la partida puede adquirir recursos, como: armas, defensas, munición o útiles.
- Poseen habilidades y ofrecen servicios: Las misiones se hacen de forma coordinada y los agentes se apoyan unos a otros para llevarlas a cabo. Acciones habituales son abrir fuego de cobertura o ayudar a otro agente a alcanzar una posición elevada a la que no podría llegar el solo.

2.7.9 Juegos de tableros

Por último mencionaremos este tipo de juegos, no por ello menos importantes, ya que fueron los primeros en plantearse como problemas a resolver mediante técnicas de búsqueda.

Juegos como el *Ajedrez,* las *damas* o *GO* fueron planteados como problemas de múltiples agentes, es decir, para una resolución mediante un algoritmo MINIMAX o una poda alfa-beta. Otro tipo de juegos, como el *8-Puzzle*, o el juego de las *Ocho reinas*, sirvieron para la aplicación de los algoritmos de búsqueda no informada.

2.8 Videojuegos y motores de Inteligencia Artificial de código abierto

En este apartado mencionaremos varias aplicaciones de código abierto. Código abierto es el término con el que se conoce al software distribuido y desarrollado libremente del que se proporciona su código fuente [von Hippel, 2001]. Hay que diferenciar este concepto del concepto de *software libre* o de *libre distribución*.

Según la Free Software Foundation, el software libre se refiere a la libertad de los usuarios para ejecutar, copiar, distribuir, estudiar, modificar el software y distribuirlo modificado. El software libre se enfoca en las libertades que les otorga a los usuarios mientras que el software de código abierto se enfoca en las ventajas de su modelo de desarrollo.

Por tanto, la idea del código abierto es la de evolucionar, mejorar y desarrollar el código fuente de un programa, aplicación, o en este caso, videojuegos. Los usuarios adaptan el código a sus necesidades y arreglan posibles fallos en el código.

En algunos juegos comerciales, pasado un tiempo su código es liberado para que los usuarios puedan observarlo, o hacer sus propios mods (modificaciones de los juegos: editan escenarios, vestimentas de los personajes, implementan misiones nuevas, etc.). En algunas páginas web, como http://liberatedgames.com/, se encuentran videojuegos como *GTA*, *Wolfstein 3D* o *Alien versus Predator*.

A continuación mencionaremos algunos videojuegos o motores que son de código abierto.

2.8.1 Screaming Racers

Screaming Racers es un videojuego de simulación de carreras donde los coches están controlados por agentes inteligentes, y en el que el objetivo es conseguir el mejor grupo de pilotos. La singularidad del juego radica en que estos agentes comienzan sin tener ningún conocimiento del entorno ni del vehículo. Los agentes tratarán de aprender y mejorar sus habilidades como pilotos, teniendo en cuenta que el único conocimiento que obtendrán es el que ellos mismos sean capaces de aprender.

La aplicación está basada en una arquitectura cliente-servidor, donde los principales componentes se encuentran dentro del servidor. Son los encargados de mantener el entorno virtual y el sistema multiagente, además de las comunicaciones entre los clientes. Estos son sus componentes:

- Motor Físico: su objetivo es el de aplicar las distintas leyes que rigen la física a todos los objetos que se encuentran en el entorno.

- Motor Gráfico: muestra en pantalla la interfaz y la apariencia de entorno.
- Agente Simulador: es el responsable de gestionar y desarrollar cada sesión de entrenamiento o competición. Debe comunicarse con todos los agentes piloto para que puedan coordinar su funcionamiento.
- Agente Controlador: el encargado de coordinar al resto de componentes.
- Agente Comunicador: implementa las tareas de comunicación entre servidor y clientes y viceversa.
- Agentes Piloto: cada uno de estos agentes implementa el cerebro de uno de los pilotos del simulador. Reciben información de sus sensores por medio del agente simulador y toman decisiones para actuar en consecuencia.

Los clientes serán utilizados por los jugadores para conectarse al servidor. El diseño multiagente proporciona una serie de ventajas en lo que se refiere a los agentes piloto. Estos pueden ser movidos desde los clientes al servidor y viceversa cuando sea necesario. Por este motivo se minimizan los problemas de la latencia en las comunicaciones.

En la siguiente figura podemos observar un ejemplo de traza donde el piloto no tiene o tiene apenas poco conocimiento (a), y una traza donde el piloto ha mejorado su conocimiento (b):

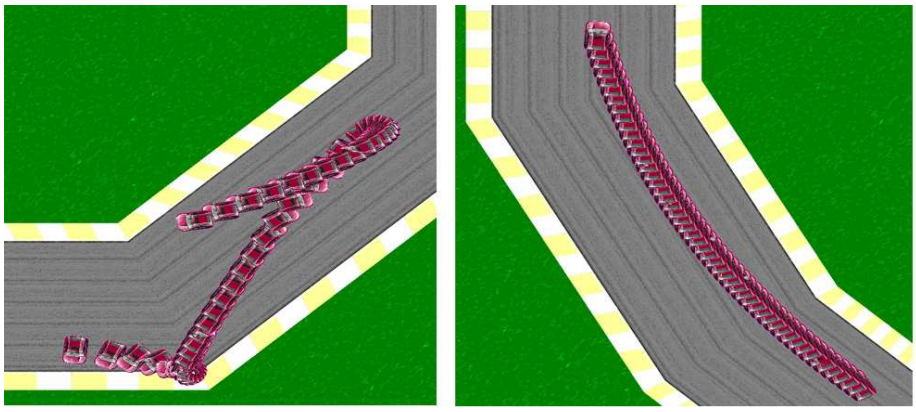

Figura 6. Ejemplos de traza en Screaming Racers.

2.8.2 ORTS

Esta aplicación no se trata realmente de un videojuego definido, sino de un motor o entorno de programación para videojuegos de *Estrategia en Tiempo Real*.
El objetivo de esta aplicación es el estudio de problemas de Inteligencia Artificial en este tipo de juegos, como la búsqueda de caminos, tratamientos con información incompleta, planificación, etc.

Este objetivo del desarrollo de este motor es debido a que la situación actual de los juegos de estrategia en tiempo real es mala, ya que es necesaria más planificación y aprendizaje en estos videojuegos para mejorarlos (los humanos son en este caso

bastante más listos que la máquina actualmente). Además se ve motivado debido a que los juegos comerciales tienen software cerrado, que impide que los investigadores de Inteligencia Artificial puedan acceder a sus módulos.

Los usuarios definen el juego que quieren desempeñar en forma de secuencias de comandos que describen todos los tipos de unidad, estructuras, y sus interacciones. Estos scripts son cargados y ejecutados por el servidor *ORTS*. La segunda parte del sistema es el software cliente que se conecta al servidor y genera acciones para los objetos del juego. El servidor envía opiniones a los clientes y recibe todas las acciones para los objetos del jugador, que luego son ejecutados. Si el cliente gráfico 3D está conectado, el mundo del juego se genera usando OpenGL y el usuario puede emitir comandos usando el ratón y el teclado.

En su página oficial[5], se encuentra toda la información, documentación, FAQs, e incluye un cuadro comparativo sobre los juegos comerciales actuales y ORTS. Se proporciona además, todo el código fuente.

Está desarrollado en C++ para los núcleos de clientes y servidores, y un lenguaje propio de scripting para definir los juegos y personalizar la interfaz gráfica.

2.8.3 Delta3D

Antes de proceder a conocer *Delta3D*, se definirá brevemente qué es un **API**. Las interfaces de programación de aplicaciones (Application programming interface, o API) son un conjunto de declaraciones sobre las funciones o procedimientos que un sistema operativo, librería, o servicio suministra para apoyar comandos hechos por programas de ordenador**.**

Los APIs que dependen de lenguaje están disponibles en un particular lenguaje de programación. Utilizan la sintaxis y los elementos de éste para lograr que el API sea lo más conveniente a ser usado en este contexto particular. Por otro lado, los APIs independientes al lenguaje están escritos de manera que puedan ser utilizados mediante varios lenguajes de programación.

Delta3D es un motor de simulación y juego de código abierto. Su diseño modular integra a otros conocidos proyectos open source, como *OpenSceneGraph*, *Open Dynamics Engine*, *Cal3D* y *OpenAL*, integrándolos así en un API sencillo y fácil de usar. En lugar de ocultar los módulos más bajos, *Delta3D* los integra a todos, permitiendo de esta manera el acceso de los programadores a los componentes más importantes. Esto proporciona un API de alto nivel que también permite al usuario hacer uso de una funcionalidad de bajo nivel, de manera opcional.

Se considera a *Delta3D* como uno de los motores más estables, y está disponible para varias plataformas diferentes, como Macintosh, Windows y Linux. El motor en sí posee gran variedad de herramientas y aplicaciones, destacando entre ellas el editor de niveles (STAGE), el editor gráfico de partículas y el visualizador en 3D.

[5] http://www.cs.ualberta.ca/~mburo/orts/

2.8.4 Multiverse

Se trata de un juego online 3D, que está compuesta de una gran red de mundos totalmente 3D creados por los usuarios por medio de un determinado software. En él se pueden crear tanto juegos o minijuegos dentro del mundo, como objetos 3D.

La tecnología de *Multiverse* es escalable, extensible y altamente personalizable, permitiendo construir un mundo virtual con multitud de objetivos, funcionalidades y servicios, y con el aspecto visual que el usuario elija. *Multiverse* posee un API de scripting del cliente basado en Python que permite acceder a mucha de las características del cliente como:

- Capacidades hardware, que pueden ser utilizadas para ajustar la configuración de la presentación grafica para obtener el mejor rendimiento en el ordenar del usuario.
- Configuración del sistema gráfico proporcionado por el motor del juego, incluyendo sombras, agua, cielo, terreno…
- La simulación del mundo, incluyendo a los jugadores, los edificios, las luces, las cámaras y otros objetos.
- Efectos 3D como las animaciones, los proyectiles y las partículas del sistema.
- Preferencias del teclado, de sonido y de los gráficos del cliente.
- Efectos coordinados, que permiten un único mensaje en el servidor para invocar un script en el cliente que implemente un efecto coordinado de audio y/o video.

El cliente, el servidor, las herramientas de desarrollo y los mundos de ejemplo son gratuitos siempre que se desarrolle con fines no comerciales. La única condición que imponen es que el mundo virtual tiene que ejecutarse en la plataforma de *Multiverse* y dentro de la red *Multiverse* (dicha red agrupa todos los mundos virtuales creados con *Multiverse*). Si se introducen anuncios en el mundo creado o se utiliza como plataforma comercial, hay que pagar licencia. Esta licencia se establece en el 10% de los beneficios brutos del mundo creado. Esta licencia está especialmente pensada para desarrolladores independientes que no pueden permitirse una gran inversión para adquirir la tecnología del mundo 3D de *Multiverse*.

2.8.5 Allegro

Allegro es una librería desarrollada en lenguaje C/C++ orientada al desarrollo de videojuegos y distribuida libremente. Funciona en plataforma Windows, DOS, o Unix. Aunque Allegro ofrece una API en lenguaje C, actualmente existen bibliotecas adicionales que permiten utilizarlo en otros lenguajes como Python, D, Lua y Pascal.

Allegro cuenta con funciones para gráficos, manipulación de imágenes, texto, sonidos, dispositivos de entrada (teclado, ratón y mandos de juego) y temporizadores,

así como rutinas de acceso al sistema de archivos. La versión más reciente de *Allegro 4* incluye soporte para el manejo de archivos de datos y una implementación por software de funciones para gráficos en 3D. La versión 5 de *Allegro*, actualmente en desarrollo, cuenta con una nueva API y cambiará la implementación por software de las rutinas gráficas por una implementación basada en OpenGL o Direct3D.

Consta de una amplia documentación y más de cuarenta ejemplos para conocer las cosas básicas e ir mejorando las habilidades. También cuenta con un foro o una lista de distribución de correo donde cualquier usuario podrá responder a las dudas de otro usuario.

En su web oficial[6] se puede observar, o descargar el código (y por supuesto editarlo), de los juegos que han sido realizados con esta librería.

2.8.6 Irrlicht

Irrlicht es un motor 3D gratuito y de código abierto, escrito en C++, el cual puede ser usado tanto en C++ como con lenguajes .Net.

Es totalmente multiplataforma, usando D3D, OpenGL y su propio software de render (renderizado de imágenes, generar imágenes desde un modelo), y tiene las características que se pueden encontrar en los motores 3D comerciales.

Se trata de un potente API de alto nivel para completar la creación de aplicaciones 2D y 3D como juegos o visualizaciones científicas. Viene con una buena documentación e integra características de representación visual como sombras dinámicas, sistemas de partículas, animación de caracteres, tanto interiores como exteriores, tecnología, y la detección de colisiones. Todo esto es accesible a través de una interfaz diseñada en C++, sencilla de usar.

Cuenta con una gran comunidad, y hay proyectos en desarrollo utilizando este motor. Se pueden encontrar mejoras para *Irrlicht*, al igual que terrenos, exportadores, tutoriales, editores, bindings (adaptaciones de bibliotecas para ser usadas en lenguajes de programación distintos a la que fueron escritas) para lenguajes Java, Perl, Ruby, Python, LUA, etc.

2.8.7 CryENGINE 2

CryENGINE 2 es un motor creado por el equipo de desarrollo del videojuego *Crysis*. Este motor cuenta con las siguientes características: edición en tiempo real, mapeado de profundidad (bump mapping), luces dinámicas, sistema de red, sistema de físicas integrado, shaders (sistema de sombreados) y música dinámicas.

Las licencias distribuyen el código fuente por completo y la documentación del motor y sus herramientas. El apoyo es proporcionado directamente del equipo de

[6] http://alleg.sourceforge.net/index.html

desarrollo que además está constantemente actualizando el motor y puede realizar talleres de enseñanza.

Aunque existen otras muchas más características (gráficas, de sonido, etc) para este motor, nos centraremos en las características relacionadas con la Inteligencia Artificial.

Búsqueda de caminos dinámica: (Dynamic Path Finding)

Los algoritmos de búsqueda avanzados en Inteligencia Artificial, permiten que las trayectorias sean modificadas en tiempo real en respuesta a los nuevos caminos que se crean o a las trayectorias existentes que se destruyen.

Objetos Elegantes: (Smart Objects)

Proporciona a los diseñadores de los niveles, una manera fácil de realizar animaciones especializadas con los objetos concretos en el nivel; así las animaciones del personaje y los objetos están correctamente alineados al comienzo y al final de las animaciones, y en la secuencia correcta de animación del juego.

Este motor está escrito enteramente en **C++**, documentado completamente y comentado, donde se puede utilizar, modificar o sustituir parcial o totalmente los componentes en nuestro motor que se requieran, para modificar requisitos particulares.

2.9 Conclusiones

Al igual que la tecnología ha ido innovando a lo largo de los años, la Inteligencia Artificial de los últimos videojuegos está sufriendo grandes mejoras; esto se debe a la aplicación de técnicas más recientes como las RNA, permitiendo alcanzar a los personajes de los juegos una inteligencia cada vez más cercana a la inteligencia humana. Otro tipo de innovaciones son también palpables en los videojuegos de última generación. Por ejemplo, en antiguos RPGs, en el mundo del juego, en ciertas ciudades o lugares siempre era de día (o de noche, pero nunca cambiaba), y esto, afectado a los personajes, siempre se encontraban esperando en la misma posición.

Actualmente, esta situación es totalmente distinta, ahora el concepto de la hora del día se hace presente, y esos personajes que antes estaban quietos ahora se mueven por sí mismos, como si tuvieran vida y tareas propias, con autonomía para realizar sus propias acciones.

Podemos concluir que la Inteligencia Artificial interviene para conseguir que los personajes NPC y *bots* tengan un comportamiento inteligente e independiente. Por ejemplo, en un videojuego del género *FPS*, es fácil de distinguir si se han aplicado técnicas de Inteligencia Artificial al personaje enemigo o no. Para este caso es muy fácil deducirlo: basta con saber que si el enemigo apenas se mueve cuando el jugador le

dispara, simplemente está programado mediante unos scripts; sin embargo, si observamos que el enemigo intenta esconderse, rodear o esquivar al jugador, en definitiva, un comportamiento autónomo para planear una estrategia, se puede deducir que el personaje tiene cierta inteligencia y que está implementado con técnicas de Inteligencia Artificial.

Por ejemplo, muchos juegos relativamente actuales del género recién mencionado, como *Halo* o *Half-Life,* son bastante lineales: aunque haya áreas extensas con varios caminos posibles, al final se unen y son muy limitados. Debido a este diseño, los programadores consiguen suplir la necesidad de tener una Inteligencia Artificial adecuada, ya que son los encargados de colocar estratégicamente a los enemigos, logran el equilibrio de la dificultad en cada nivel, y aplican scripts para situaciones especiales. Este tipo de combinaciones, hacen que no sea necesaria una Inteligencia Artificial para tomar "decisiones tácticas", pero sí que sería conveniente reducir la carga de trabajo de los diseñadores y mejorar la jugabilidad.

Aunque no exista una Inteligencia Artificial verdaderamente implementada en ciertos videojuegos, dado el nivel que están adquiriendo éstos en la actualidad, sí que se debería apostar más por utilizar algunas de las técnicas mencionadas antes, ya que como hemos explicado otorga gran riqueza a los videojuegos. Es una tecnología que está en crecimiento y se le debe prestar una destacada importancia. Además, el jugador con el tiempo va a exigir más calidad en los videojuegos, no sólo gráficamente, porque es algo que va implícito en la evolución de los videojuegos, sino un nivel de Inteligencia Artificial mejor, ya que el deseo del jugador es del de poder jugar contra la máquina como si fuera otro jugador humano.

3 OBJETIVOS DEL PROYECTO FIN DE CARRERA

El objetivo principal de este Proyecto de Fin de Carrera es desarrollar e implementar un cliente que convierta al juego AI-LIVE en un sistema que permita que todas las acciones implementadas que realiza un actor o varios actores dentro de un escenario del juego tengan un sentido. Es decir, asemejar el juego a la simulación de la vida real. Esto es la realización de una secuencia lógica de acciones, como por ejemplo: que un actor trabaje, y después de ello coma, se duche, descanse, y después cuando tenga tiempo libre emplear ese tiempo en jugar, leer, comunicarse con otros actores, etc.

Anteriormente en el juego AI-LIVE las acciones se iban realizando de manera que satisfacían las necesidades básicas que iban surgiendo en cada turno, llegando un punto en el que el actor no tenía nada más que hacer y se dedicaba a moverse de una casilla a otra del escenario. Esto es lo que se pretende que no pase, teniendo un objetivo que cumplir.

Para ello se ha pensado en varias ideas. Una de ellas es la idea del **tiempo**. En versiones anteriores, en el juego todas las acciones duraban lo mismo, siendo algo irreal ya que una persona no duerme el mismo tiempo que utiliza en comer. Para ello se ajustarán las acciones en el tiempo real que se suele utilizar en cada una de ellas. Ya que el tiempo en AI-LIVE transcurre por turnos, la medición del tiempo de las acciones vendrá dada por el número de turnos que dure cada una.

La forma en que se implementan las necesidades básicas de los actores es por medio de unas variables denominadas "drives" que pueden tomar valores positivos. El valor 0 representa que la necesidad a la que se refiere dicho drive está cubierta. Por cada necesidad básica considerada se define un drive y acciones que permitan a los actores disminuir dicho drive. Por ejemplo, para la necesidad básica de comer se define el drive hambre y la acción de comer, o para la necesidad de descansar se define el drive cansancio y acciones como dormir o sentarse en el sofá para satisfacerla. Con la inclusión del concepto del tiempo, será necesario realizar el ajuste de la modificación de los valores de los drives del actor en cada turno. Lógicamente, si la acción de dormir en un turno disminuía un valor de cansancio, al modificarlo durante más turnos hay que disminuir un valor equivalente al número de turnos que dura la acción de dormir. También cabe la posibilidad de que el actor pueda interrumpir su acción para realizar otra.

Lógicamente, si la acción de dormir en un turno disminuía un valor de cansancio, al modificarlo durante más turnos hay que disminuir un valor equivalente al número de turnos que dura la acción de dormir. También cabe la posibilidad de que el actor, tanto manual como por la Inteligencia Artificial, pueda interrumpir su acción para realizar otra.

Otro de los factores a contemplar es que una acción supone una modificación de los valores de otros drives del actor además del que trata el objeto o la acción en sí.

Es decir, si un actor está jugando, disminuye su aburrimiento, pero aumentan su cansancio, suciedad, etc. Esto supone otra manera más de que los drives del actor aumenten para poder realizar otras acciones.

Para conseguir un mayor realismo en el comportamiento de los actores cuando interactúan varios en un mismo escenario, se incorporará el modelo emocional definido en las primeras versiones de AI-LIVE, dentro del proceso de toma de decisiones de los actores. Inicialmente, el modelo emocional solo afectaba a las acciones de tipo comunicativas, en particular, la acción de contar un gusto sobre un objeto particular.

4 MEMORIA-TRABAJO REALIZADO

4.1 Introducción

En este apartado de la memoria se describe el trabajo realizado en el sistema AI-LIVE, detallando la implementación propuesta para alcanzar los objetivos del proyecto. El trabajo de este proyecto se ha realizado con otro alumno de forma conjunta, encargándose del lado del servidor de la arquitectura.

Al comenzar el proyecto disponíamos de un sistema en el que un actor era capaz de realizar varias acciones correspondientes a la satisfacción de necesidades físicas básicas (comer, cuidar la higiene, dormir, etc.). El objetivo del juego era conseguir el mayor grado de felicidad del actor medido en función de la consecución de todas sus necesidades. Estas necesidades están representadas por unas variables o "drives", que comprenden un rango de valores. Cuanto mayor es el valor del drive, más importante es subsanar esta necesidad. Para poder satisfacer estas necesidades, los actores se mueven por un escenario ejecutando acciones que disminuyen el valor de un único drive. Cada acción precisa de la utilización de un objeto en particular ubicado en un lugar del escenario. Por ejemplo, para satisfacer el hambre el actor puede comer y para ello necesita tener algún alimento que estará en la cocina. O para satisfacer el nivel de higiene, se puede lavar utilizando la ducha que estará en el cuarto de baño.

Por otro lado, existía un motor emocional implementado [Jiménez, 2008] mediante el cual un actor era capaz de comunicarse con otro, expresando sus gustos por ciertos objetos, según el estado de ánimo o personalidad entre otros factores. El estado emocional del actor sólo se veía afectado al hablar con otro actor sobre sus gustos. Este módulo no estaba integrado en la aplicación y contenía únicamente las acciones de tipo comunicativas entre actores.

El trabajo realizado ha sido el de dotar al sistema AI-LIVE de una mayor complejidad para conseguir mayor realismo conjuntando las características antes mencionadas y otras nuevas. Esto se ha conseguido logrando una mayor interactividad entre el actor y los objetos del escenario del juego. Anteriormente el actor disponía de

una bolsa que contenía ciertos objetos que usaba al momento de realizar alguna acción donde fueran necesarios. Se ha eliminado esta bolsa, obligando a los actores a localizar los objetos que necesitan para realizar una acción, permitiendo llevarlos consigo, o teniendo que dejarlos en el suelo en caso de no poder cargar con más objetos.

Se incorporó el módulo que gestiona las emociones de los actores y las relaciones entre ellos en todo el proceso de toma de decisiones, en vez de la limitación del sistema anterior a las acciones comunicativas verbales. Combinando este modelo se obtiene una mayor riqueza en cuanto a la interacción con otros actores y no solo con objetos como estaba antes.

Otro de los conceptos para dotar al sistema de mayor complejidad es el del **tiempo**. Anteriormente todas las acciones tenían la misma duración (un turno), por lo que parecía poco realista, ya que por ejemplo no se emplea el mismo tiempo en dormir que en comer. Para ello se diseñó un modelo que aunara criterios entre los drives de los actores y la forma en la que el paso del tiempo fuera aumentando la necesidad de cubrir estos drives.

Se ha implementado también que el conjunto de los valores de las necesidades básicas de los actores influya en su estado anímico, notándose por ejemplo a la hora de comunicarse con otro actor.

Por último, se ha realizado un script de ejecución que permita automatizar la ejecución del juego AI-LIVE, personalizando distintas variables y parámetros de una manera muy sencilla lanzando servidor y tantos clientes como se desee.

Es necesario indicar que para comentar la funcionalidad implementada, se mencionarán ciertos aspectos implementados en el servidor que se complementan con el cliente para conseguir dicha funcionalidad. El servidor fue desarrollado paralelamente a este proyecto por otro compañero, coordinándose ambos proyectos.

4.2 Arquitectura de la aplicación

El proyecto AI-LIVE se basa en una arquitectura cliente-servidor, donde la comunicación entre éstos se realiza mediante *sockets*, sobre TCP/IP.

Este tipo de estructura favorece el reparto de la capacidad de proceso entre los clientes y servidores, así como la organización debida a la centralización de la gestión de la información y separación de responsabilidades.

En nuestra aplicación, cada cliente representa un actor o personaje que habita en un escenario, y que es capaz de decidir su propio comportamiento según las variables del entorno. Una vez realizada la acción, el servidor actualiza los valores del escenario, objetos y características de los actores.

Estos clientes son generalmente de Inteligencia Artificial, pero se han implementado otros, como clientes manuales, en los que un usuario es el que decide la acción del actor a realizar; o clientes gráficos (GUI) para representar el escenario del juego.

El servidor se encarga de recibir la acción que desea realizar cada cliente, comprobando su factibilidad y actualizando el entorno de acuerdo a los efectos de dicha acción para que el siguiente cliente que tiene el turno (asignado por el servidor) reciba el nuevo estado.

La Figura 7 representa un esquema de la arquitectura de AI-LIVE y las siguientes secciones describen en detalle cada componente.

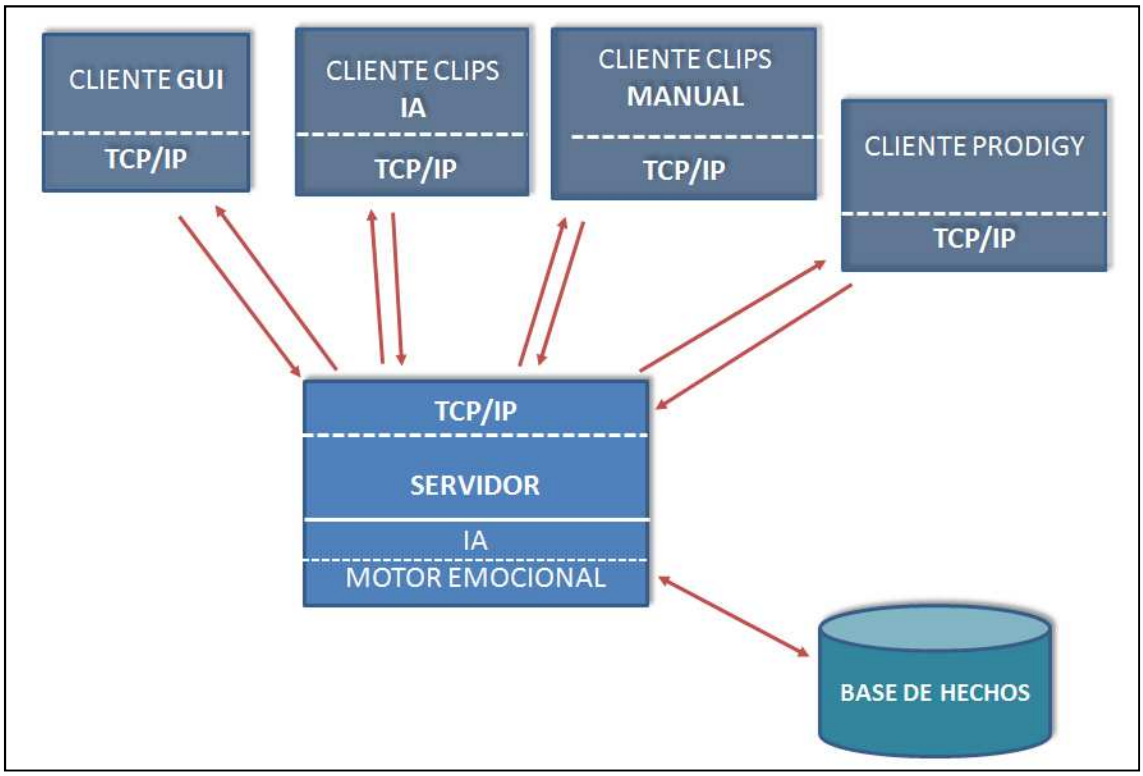

Figura 7. Arquitectura Cliente-Servidor.

4.2.1. Servidor

El servidor contiene la información del estado del universo y se encarga de ejecutar las acciones correspondientes solicitadas por los clientes. Además, se encarga de establecer los protocolos de conexión de los clientes y de gestionar posteriormente el sistema de turnos.

Existen tres partes diferenciadas en el servidor:

- Módulo principal (*server.c*): se encarga de gestionar las conexiones, de establecer los turnos y de las llamadas al módulo de Inteligencia Artificial.

- Módulo de Inteligencia Artificial (*server.clp*): contiene el estado del universo y ejecuta las acciones recibidas de los clientes, modificando el estado.

- Módulo del motor emocional (*emotional_engine.c*): contiene las funciones encargadas de actualizar el estado de los agentes y su relación cuando tiene lugar una acción comunicativa. Se invoca desde el módulo de Inteligencia Artificial.

4.2.2. Clientes

Los clientes se encargan de generar las peticiones que se envían al servidor. Existen dos partes claramente diferenciadas en los clientes:

- Módulo principal: se encarga de establecer la conexión con el servidor y de llevar a cabo el protocolo de comunicaciones.

- Módulo de ejecución: realiza la funcionalidad propia del cliente, según el tipo que sea. Esta funcionalidad se especificará a continuación.

4.2.2.1. Tipos de clientes

- Cliente de CLIPS: Cliente con motor de Inteligencia Artificial, que integra un sistema de producción basado en reglas, CLIPS, y controla a un personaje dentro del universo de realidad simulada.

- Cliente de CLIPS manual: Cliente con el motor de Inteligencia Artificial al igual que el anterior, con la diferencia de que antes de efectuar la acción, muestra por pantalla una lista de acciones disponibles en las que el usuario será el que decida, controlando por tanto a un personaje manualmente dentro del universo de juego.

- Cliente de Prodigy: Cliente con motor de Inteligencia Artificial, que integra un planificador de tareas, Prodigy, y controla a un personaje dentro del universo de realidad simulada.

- Cliente GUI: Se trata de un cliente de interfaz gráfica de usuario con una representación en 3D del universo del juego.

4.2.3 Protocolo de comunicación

En este apartado se detallará el mecanismo de comunicación entre servidor y clientes para el intercambio de información, entendiendo su funcionamiento.

Para el intercambio de mensajes, se establece un protocolo de comunicación que utiliza etiquetas, las cuales se nombran a continuación:

- **HOLA**: utilizada tanto por el servidor como los clientes para iniciar la conexión. En él se indica la versión y las diversas opciones soportadas por los elementos que intervienen en la comunicación.

- **STAG**: utilizada por los clientes para indicar el escenario donde comenzarán.

- **ACTR**: utilizada por los clientes para especificar el identificador del actor que están controlando.

- **ACTN**: utilizada por los clientes para indicar la acción que han decidido, para que la ejecute el servidor.

- **STAT**: utilizada por el servidor para enviar el estado del juego a un cliente.

- **GO**: utilizada por el cliente GUI implementado y el servidor para indicar que el módulo actual ha terminado la actualización de la información y que el otro módulo puede continuar su ejecución.

- **EX**: utilizada por el cliente GUI implementado para indicarle al servidor que ha finalizado su ejecución.

A continuación se describe el conjunto de pasos a seguir, tanto por el servidor como por los clientes, para establecer y mantener una conexión entre éstos a través del protocolo de comunicación.

4.2.1.1. Pasos a realizar por el servidor

1. Iniciar el motor de Inteligencia Artificial del servidor.

2. Iniciar el escuchador de red.

3. Conectarse con el cliente, cuando este lo solicita, a través del protocolo establecido.

4. Bucle principal en ejecución:

    - Seleccionar al cliente que tenga asignado el turno.

    - Enviar el estado al cliente actual y al GUI.

    - Recibir la acción del cliente.

    - Enviar la acción al servidor de Inteligencia Artificial.

    - Recibir el estado de Inteligencia Artificial.

4.2.3.2. Pasos a realizar por los clientes de Inteligencia Artificial y manual

1. Conectarse al servidor.

2. Bucle principal en ejecución:

    - Recibir el estado completo.

    - Interpretar el estado recibido.

    - Elegir la acción a realizar (de distinta forma según el tipo de cliente)

- Enviar la acción a realizar.

4.2.3.3. Pasos a realizar por el cliente GUI

1. Conectarse al servidor.
2. Bucle principal durante la ejecución:
    - Recibir el estado referente a los objetos gráficos.
    - Representar el estado recibido.

4.2.3.4. Inicio de la conexión

El cliente se conecta al servidor de escucha. El servidor puede cerrar la conexión en cualquier momento si el cliente no es adecuado para él.

1. El servidor envía al cliente **HOLA*xxyz***. Se utiliza este comando para iniciar la conexión con el cliente:

    - ***xx***: indica la versión del servidor.
    - ***y***: opciones que soporta el servidor (todos los *flags* de tipo cliente soportados).
    - ***z***: reservado (actualmente es 0).

2. El cliente envía al servidor **HOLA*xxyz***. Este comando se utiliza para iniciar la conexión con el servidor, utilizando la etiqueta HOLA y los parámetros siguientes:

    - ***xx***: indica la versión del cliente.
    - ***y***: opciones que soporta el cliente (tipo de cliente).
    - ***z***: reservado (actualmente es 0).

3. El cliente envía al servidor **STAG*xxxxstage*0**. Con la etiqueta STAG se envía el escenario elegido por el cliente:

    - ***xxxx*:** entero de 32 bits que indica la longitud en bytes del escenario enviado.
    - ***stage*:** identificador del escenario.
    - El comando termina con un byte 0, que no se tiene en cuenta.

4. Sólo los clientes de Inteligencia Artificial y manual envían al servidor **ACTRxxxxactor0**. Este comando se utiliza para el envío del identificador del actor controlado por el cliente:

    - ***xxxx***: entero de 32 bits que indica la longitud en bytes del identificador del actor.
    - ***actor***: identificador del actor generado aleatoriamente.
    - El comando termina con un byte 0, que no se tiene en cuenta.

5. Los clientes de Inteligencia Artificial y manual envían al servidor **ACTN*xxxxprofile*(id[*a_id*])(stage [*stg*])0**. Este comando se utiliza para el envío del perfil del actor:

- ***xxxx***: entero de 32 bits que indica la longitud en bytes del perfil del actor.
- ***profile***: perfil del actor que se envía al servidor.
- ***a_id***: identificador del actor.
- ***stg***: escenario en el que está el actor.
- El comando termina con un byte 0, que no se tiene en cuenta.

4.2.3.5. Bucle de la conexión

1. El servidor envía al cliente **STAT*xxxxstate*0**. Con esta etiqueta se envía el estado del juego al cliente. Los siguientes parámetros son:

- ***xxxx***: entero de 32 bits que indica la longitud en bytes del estado enviado.
- ***state***: estado enviado del servidor al cliente.
- El comando termina con un byte 0, que no se tiene en cuenta.

2. El cliente responde al servidor. Distinguimos dos casos:

a. Si se trata de un cliente de Inteligencia Artificial o manual, éste responde con **ACTN*xxxxaction*0**. Este comando se utiliza para el envío de la acción del cliente al servidor:

   - ***xxxx*:** entero de 32 bits que indica la longitud en bytes de la acción enviada.
   - ***action*:** acción enviada del cliente al servidor.
   - El comando termina con un byte 0, que no se tiene en cuenta.

b. Si se trata del cliente GUI, éste responde con **GO** para indicarle al servidor que ha terminado de representar el estado gráficamente y que continúe con su ejecución, o **EX** para indicarle que ha finalizado su ejecución y que no le envíe el estado en próximas iteraciones.

4.3 Modelo de conocimiento

El modelo que vamos a representar es una visión gráfica de la ontología del sistema, la cual recoge todas las clases que existen en AI-LIVE, así como sus atributos y relaciones entre las distintas clases. La ontología es la parte fundamental del sistema, ya que cualquier modificación debe recogerse en ella. Es común tanto para el servidor como para los distintos clientes.

Para una mejor comprensión, en la representación del siguiente diagrama figurarán únicamente las clases de la ontología, sin sus atributos. Estos serán especificados y explicados en la posterior descripción detallada de todas las clases.

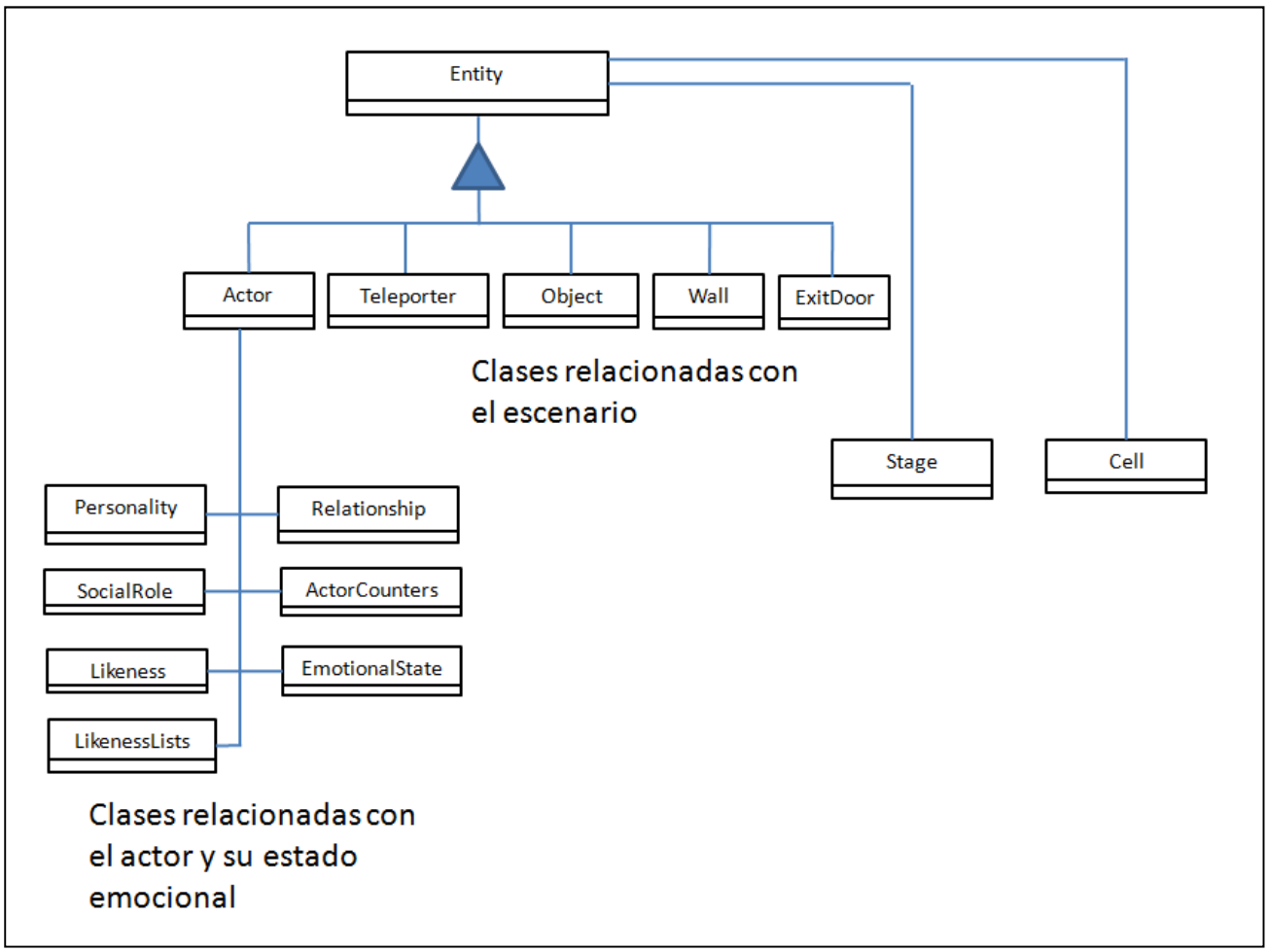

Figura 8. Modelo de Conocimiento, Clases principales.

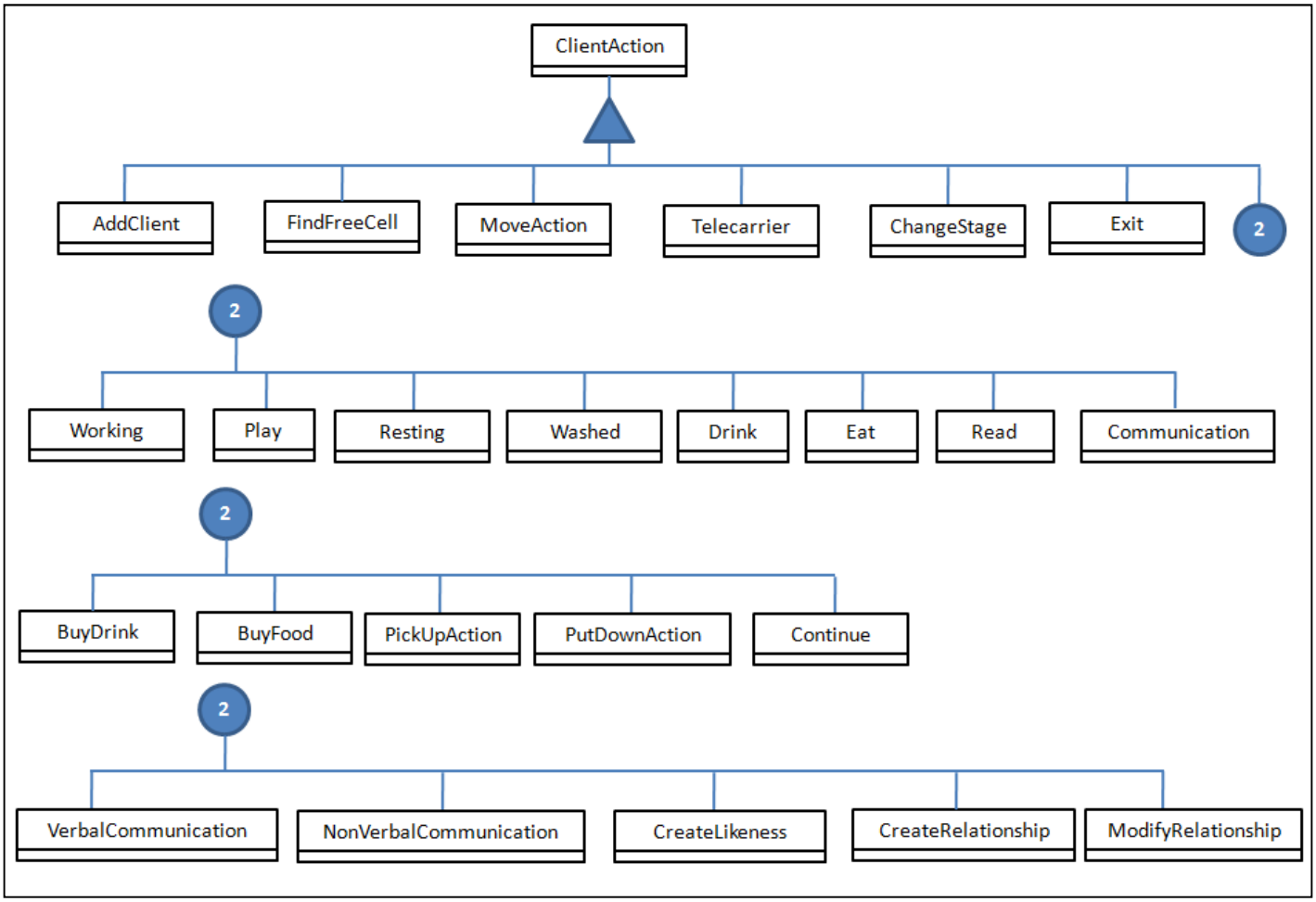

Figura 9. Modelo de Conocimiento, Subclases de ClientAction.

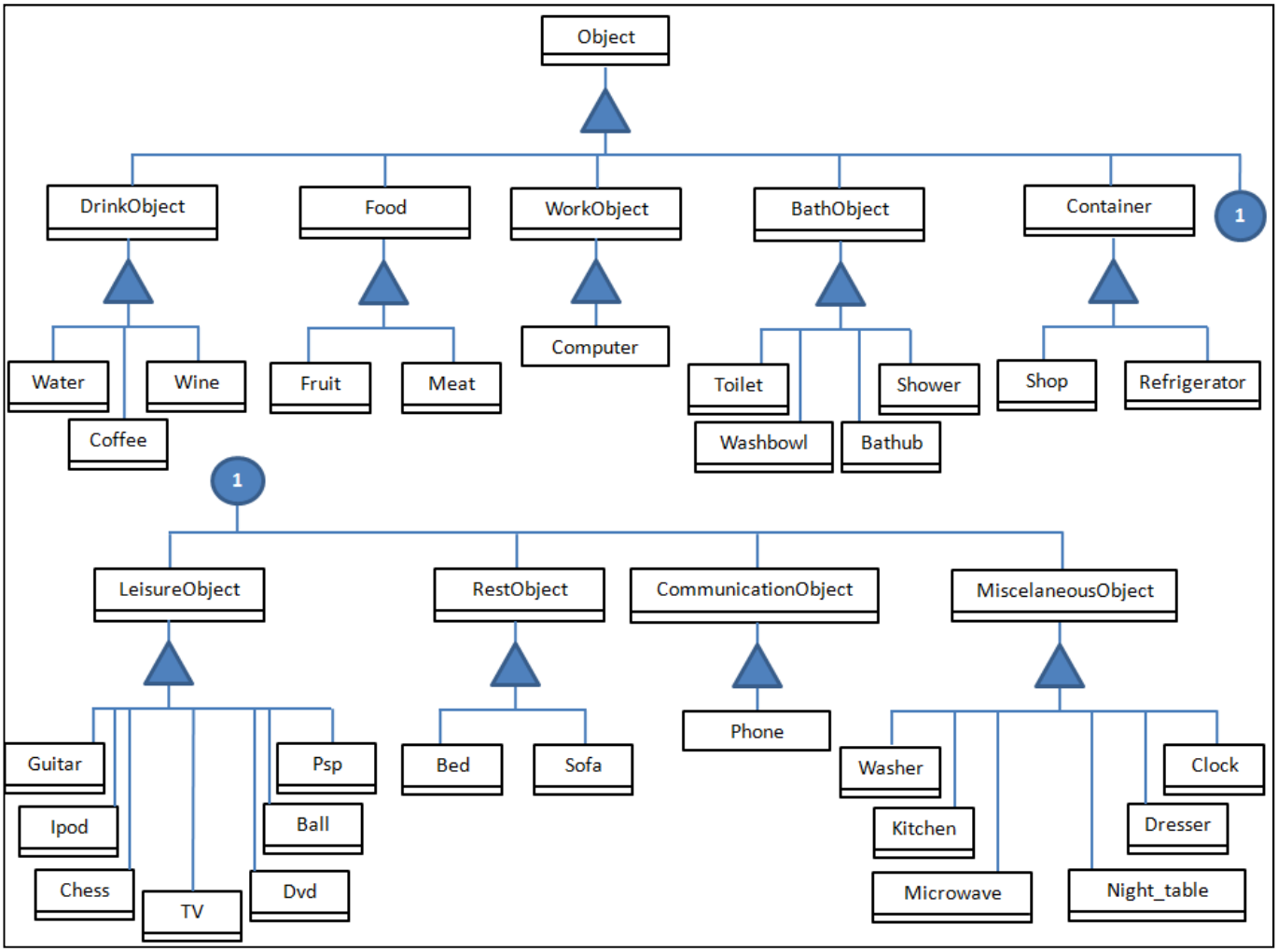

Figura 10. Modelo de Conocimiento, Subclases de Object.

A continuación se describen cada una de las clases mostradas gráficamente anteriormente. Primero se explica la clase Entity, la cual es una generalización de todas las entidades existentes en AI-LIVE. Se explicará junto con las clases que representan los distintos elementos que constituyen el escenario.

Después se explicará la clase Actor, una de las entidades más importantes del juego, junto con aquellas clases que se relacionan directamente con ésta (estado emocional, etc).

Por último distinguiremos otros dos bloques de clases. Uno de ellos se corresponde con aquellas clases que representan las distintas acciones que puede realizar un actor. El otro bloque de clases corresponde con aquellas que representan los diferentes tipos de objetos que existen en el universo AI-LIVE.

Entity: Es una generalización abstracta de todos los tipos de entidades que existen en el juego. Contiene los atributos comunes a todas ellas. Los atributos que componen esta clase son:

- **stage**: escenario al que pertenece la entidad.
- **name_**: nombre de la entidad.
- **x**: coordenada X de la entidad dentro del escenario en el que se encuentra.
- **y**: coordenada Y de la entidad dentro del escenario en el que se encuentra.
- **z**: coordenada Z de la entidad dentro del escenario en el que se encuentra.
- **sizeX**: tamaño respecto al eje X.
- **sizeY**: tamaño respecto al eje Y.
- **sizeZ**: tamaño respecto al eje Z.
- **angle**: orientación de la entidad dentro del escenario. Los posibles valores que pueden tomar son: north, south, east, west.
- **weight**: peso de la entidad.
- **volume**: volumen de la entidad.
- **entityGraphic**: nombre del fichero *.mesh* asociado a la entidad para poder representarla en la interfaz gráfica.
- **entityState**: estado de la entidad utilizado para representarla en la interfaz gráfica.
- **in**: instancia de una entidad (puede ser tanto un actor, como un objeto contenedor) que contiene a esta entidad. En caso de no estar contenido en ninguna entidad su valor será nil.
- **accessCell**: instancia de la celda de acceso de la entidad. La mayoría de objetos y entidades tienen una celda de acceso. Los actores por ejemplo, tendrán este valor a nil.

Stage: clase que representa un escenario del juego. Un escenario está formado por **celdas** (clase **Cell**, que veremos más adelante). Los atributos que contiene corresponden con todos los elementos y características que tiene el escenario:

- **entities**: lista de instancias de entidades que forman el escenario.
- **name_**: nombre del escenario.
- **length**: longitud del escenario expresada en número de celdas.
- **width**: anchura del escenario expresada en número de celdas.
- **heigth**: altura del escenario expresada en número de celdas.
- **stageGraphic**: textura para representar el escenario en la interfaz gráfica.

Cell: Clase que recoge la información relativa a una celda de un escenario del juego. Representa la porción mínima e indivisible del escenario. Se utiliza también en la detección de colisiones, controlando que la celda esté ocupada o como posición de acceso de ciertos objetos. Sus atributos son:

- **stage**: escenario al que pertenece la celda.
- **inCell**: en caso de que la celda no esté ocupada el valor del atributo será nil, en otro caso, contendrá la instancia de la entidad situada en ésta.
- **occupied**: indica si la celda está ocupada o libre.
- **x**: coordenada X de la celda dentro del escenario.
- **y**: coordenada Y de la celda dentro del escenario.
- **z**: coordenada Z de la celda dentro del escenario.

ExitDoor: clase que representa un acceso para salir del escenario y terminar la ejecución del actor. Para los clientes CLIPS de Inteligencia Artificial, el fin de la ejecución se realiza cuando el usuario pulsa Ctrl+C. Se captura la señal y el cliente envía una acción de tipo *Exit* automáticamente al servidor. Por tanto para estos clientes no es necesario un objeto como una puerta de acceso. Este acceso está contemplado para una ejecución con un cliente manual dirigido por el usuario, ya que para que la regla cumpla las precondiciones de la acción *Exit*, una de ellas es la existencia de una puerta de salida, permitiendo al usuario la opción de decidir realizar dicha acción.

No contiene atributos propios, pero utiliza en especial el atributo accessCell, heredado de Entity, para localizar la celda donde se sitúa.

Wall: Clase utilizada para poder representar gráficamente las paredes del escenario, y para delimitar el escenario en sí. No consta de atributos propios. Los atributos heredados de la clase Entity sirven para establecer el tamaño, altura o grosor de las paredes. Los actores no pueden atravesar las paredes del escenario.

Teleporter: clase que representa un acceso por el cual se puede cambiar de escenario. Para poder cambiar de un escenario a otro se debe localizar la celda de acceso a esta entidad. Es necesario que haya uno como mínimo para permitir el cambio de escenario. Consta de un atributo propio:

- **destinationStage**: instancia del escenario al que se cambia el actor.

Actor: clase que representa a un actor en el universo AI-LIVE. Todos los atributos que conforman la clase se detallan a continuación:

- **gender**: sexo del actor (masculino o femenino).

- **welfare**: nivel de bienestar del actor. Se calcula a través de los atributos hunger, thirst, solitude, tiredness, dirtness y boredom.

- **turn**: atributo utilizado para controlar las acciones que duran más de un turno, indicando los turnos que quedan.

- **continueProbability**: probabilidad para determinar si continúa con una acción que dura varios turnos.

- **readedBooks**: lista de instancias de libros leídos por el actor. Sirve para que no lea el mismo libro.

- **target**: atributo que recoge la instancia de un objeto concreto que ha localizado para satisfacer una necesidad básica.

- **usedLoad**: peso total actual de los objetos que lleva cogidos el actor.

- **maxLoad**: peso máximo que puede soportar el actor para llevar objetos.

- **catchedObjects**: lista de los objetos que lleva encima el actor.

- **personality**: instancia de la clase Personality, que contiene los atributos sobre la personalidad del actor.

- **social_role**: instancia de la clase SocialRole, que contiene los atributos sobre el estatus social del actor.

- **emotions**: instancia de la clase EmotionalState, que contiene los atributos sobre el estado emocional del actor.

- **counters**: instancia de la clase ActorCounters. Contiene los valores de los drives de necesidades básicas del actor.

- **maxCounters**: instancia de la clase ActorCounters que contiene los valores de los máximos de los drives, a partir de los cuales se activan las reglas que satisfacen las necesidades básicas correspondientes.

- **likeness_lists**: instancia de la clase LikenessLists, que contiene las listas de instancias de gustos por objetos, acciones, actores o ideas.

SocialRole: Representa el estado social del actor. Los atributos de esta clase son:

- **age**: edad del actor.

- **sex**: género del actor (0 masculino, 1 femenino).

- **status**: indica el papel que representa una persona en la sociedad que se está recreando durante una ejecución. Se plantea como una suma de diferentes puntuaciones según su papel social. Así, por ejemplo, si representamos relaciones familiares, un padre tiene mayor estatus que su hijo, y más aún que su nieto.

- **actor**: instancia del actor al que pertenece la instancia de esta clase.

EmotionalState: Clase que representa el estado emocional vinculado a un actor. Los valores que toman los dos atributos que contiene esta clase se identifican como una emoción particular del actor (alegría, tristeza, miedo, agitación, etc.). También se utiliza en las clases Likeness y Relationship para representar el sentimiento que produce en el actor un objeto o una persona, según se describirá más adelante.

Estos atributos toman valores continuos de un rango desde -1 a 1.

- **valence**: valencia de la emoción, nivel de satisfacción o agrado.

- **arousal**: activación del actor, nivel de actividad.

- **actor**: instancia del actor al que pertenece la instancia de esta clase.

Personality: Clase que recoge los cinco factores que componen la personalidad de un agente. Sus atributos pueden tomar valores continuos entre 0 (si el agente no tiene rasgos característicos de ese factor) y 1 (si tiene todas las características asociadas a ese factor).

- **agreeableness**: afabilidad del actor.

- **conscientouness**: meticulosidad o autodisciplina del actor.

- **extraversion**: extroversión del actor.

- **neuroticism**: nivel de volubilidad del actor.

- **openness**: nivel de mentalidad abierta.

- **actor**: instancia del actor al que pertenece la instancia de esta clase.

ActorCounters: Representa los contadores de las necesidades básicas del actor.

- **hunger**: variable que indica el nivel de hambre.

- **thirst**: contador que indica el nivel de sed.

- **dirtness**: atributo correspondiente a la suciedad del actor.

- **tiredness**: contador de cansancio del actor.

- **boredom**: variable que indica el aburrimiento del actor.

- **solitude**: variable que indica el nivel de soledad del actor.

- **wealth**: contador de la riqueza (dinero que tiene el actor para comprar).

- **actor**: instancia del actor del que se trata.

- **type**: tipo de contador. Se contemplan tres casos:

- tipo *Actor*, para indicar que la instancia de esta clase se refiere a los valores actuales de los drives del actor, en los cuales cuanto mayor es su valor, más importancia requiere satisfacer la necesidad.
- tipo *Max*, para indicar que la instancia de la clase recoge los valores máximos (que son fijos) a partir de los cuales el actor considera que tiene que satisfacer la necesidad correspondiente.
- tipo *Continue*, para indicar que la instancia contiene los valores con los incrementos o decrementos de los drives cuando una acción dura más de un turno.

Likeness: Clase que representa un vínculo o gusto entre un agente con objetos, acciones, ideas u otros agentes del universo virtual. Los atributos de esta clase son:

- **source**: actor al que pertenece el gusto.

- **destination**: objeto, acción o idea que le gusta al actor.

- **startTime**: momento en que se crea el gusto.

- **greater:** indica cuál es el mayor gusto por el objeto.

- **emotionalState**: instancia de la clase EmotionalState para indicar las emociones que despierta en el actor el objeto del gusto.

LikenessLists: clase que reúne las distintas listas de gustos del actor hacia objetos, acciones o ideas. Cada atributo corresponde a una lista de gustos. Estas listas, por tanto, contienen instancias de la clase **Likeness** comentada anteriormente.

- **likenessList**: lista de objetos que le gustan al actor.

- **inter_likeList:** lista de gustos intermedia y positiva.

- **neutrallikenessList**: lista de instancias de gustos neutrales del actor.

- **inter_dislikeList**: lista de gustos intermedios y negativos.

- **dislikenessList**: lista de objectos, acciones, etc, que menos le gustan al actor.

- **actor**: instancia del actor al que pertenecerá la instancia de esta clase.

Relationship: Clase que representa una relación social unidireccional de un agente (origen) con otro agente (destino).

- **source**: actor origen de la relación.

- **destination**: representa al actor destino de la relación.

- **kind**: tipo de la relación entre los actores (por edad, estatus social o parentesco).

- **strength**: fuerza de la relación entre los actores.

- **dominance**: grado de dominancia del actor origen sobre el actor destino.

- **startTime**: momento en el que se inicia la relación.

- **emotionalState**: instancia de la clase EmotionalState, que representa las emociones que despierta el actor destino en el actor origen.

PrintMyStuff: Especialización de la clase **ServerAction**, que sirve para imprimir la acción que realiza un actor, tanto en el terminal como en el fichero de traza que se genera. Contiene un atributo que es el siguiente:

- **destination:** acción que realiza el actor.

El siguiente grupo de clases forman el conjunto de todas las acciones que puede realizar un actor en el universo AI-LIVE. La clase padre común a todas ellas es la clase **ClientAction**, que comentaremos a continuación.

ClientAction: Clase que representa una acción realizada por un actor. Todas las acciones que se implementen posteriormente deben ser heredadas de esta clase. Sus atributos propios, y heredados por tanto por el resto de subclases, son los siguientes:

- **stage**: indica el escenario en el que se realiza la acción. El atributo es una instancia de ese escenario.

- **actor**: instancia del actor que realiza la acción.

AddClient: Esta acción se ejecuta una única vez por cada cliente y, a diferencia de las restantes acciones, no responde a una decisión de actuación tomada por el motor de Inteligencia Artificial del cliente, sino que es parte del proceso de inicialización. Cuando se establece la conexión de un cliente a la aplicación se crea una instancia de la clase **Actor** a partir de los valores de una instancia de **AddClient**, que debe existir previamente en un fichero conocido como perfil del cliente. Los atributos de esta clase corresponden a los atributos de la clase **Actor** que se desean configurar inicialmente (por ejemplo coordenadas iniciales en el escenario, entidad gráfica que lo representa, valores de personalidad, de su estado emocional, etc.). El resto de atributos de la clase **Actor** que no estén especificados en esta clase **AddClient** tomarán valores aleatorios (como por ejemplo los valores iniicales de los drives de necesidades básicas), o el valor por defecto definido en la ontología.

- **id**: corresponde con el identificador del nuevo actor a añadir al escenario. Este identificador es único y es una instancia del id del actor.

- **angle:** orientación del actor. Por defecto es “norte”.

- **gender**: género del actor.

- **maxLoad:** este atributo define el peso máximo total de objetos que puede soportar el actor para llevarlos encima.

- **maxVolume**: indica el volumen máximo de objetos que puede llevar el actor.

- **name_**: nombre del actor. No confundir con su id.

- **entityGraphic**: nombre del fichero gráfico asociado al actor que sirve para representarlo en la interfaz.

- **sizeX**: tamaño del actor respecto del eje X.

- **sizeY**: tamaño del actor respecto del eje Y.

- **sizeZ**: tamaño del actor respecto del eje Z.

- **volume**: volumen del actor.

- **weight**: peso del actor.

- **maxhunger**: valor de hambre en el que se activan las reglas que permiten satisfacer la necesidad.

- **maxthirst**: valor de sed en el que se activan las reglas que permiten satisfacer la necesidad.

- **maxdirtness**: valor de suciedad en el que las acciones para reducir la suciedad se activan por ser necesarias.

- **maxtiredness**: valor de cansancio a partir del cual se activan las acciones para descansar.

- **maxboredom**: valor de aburrimiento del actor en el que se activan las reglas para disminuir el aburrimiento.

- **maxsolitude**: valor máximo de soledad que puede tener el actor sin que las acciones estén activas para reducir la soledad.

- **minwealth**: valor mínimo de dinero del actor para que se activen las acciones necesarias para ganar dinero (trabajando).

- **continueProbability**: valor que sirve para calcular la probabilidad de que el actor continúe o deje una acción que dure varios turnos.

- **agreeableness**: propiedad de afabilidad del actor.

- **conscientiousness**: nivel de escrupulosidad del actor.

- **extraversion**: valor de extroversión del actor.

- **neuroticism**: valor de neuroticismo del actor.

- **openness**: valor que indica si el actor es más o menos “abierto” (receptivo).

- **age**: edad.

- **sex**: género del actor.

- **status**: estatus social del actor.

- **valence**: valencia emocional del actor.

- **arousal**: activación emocional del actor.

- **likenessList**: lista de objetos que le gustan al actor.

- **dislikenessList**: lista de objetos que le disgustan al actor.

- **neutrallikeList**: lista de objetos con gusto neutral.

- **inter_likeList**: lista intermedia de objetos que le gustan al actor.

- **inter_dislikeList**: lista intermedia de objetos que le disgustan al actor.

CreateLikeness: especialización de la clase **ClientAction**, consistente en la creación del gusto (una instancia de la clase Likeness) por parte del actor por un objeto, acción o idea. Se ejecuta cuando se conecta un cliente.

- **destination**: objeto del cual creamos el gusto.

- **source**: actor del cual se crea el gusto.

- **valence**: valencia emocional del gusto.

- **arousal**: activación emocional del gusto. Junto con la valencia del gusto se crea posteriormente una instancia de la clase EmotionalState.

CreateRelationship: representa la creación explícita (según la clase) de una relación entre dos actores. Los atributos propios que componen esta clase son:

- **destination**: actor destino al que va dirigida la relación.

- **source**: actor origen de la relación.

- **kind**: tipo de relación que tiene lugar entre los dos actores. Puede ser basada en edad (AgeBased), estatus social (StatusBased) o parentesco (BloodBased).

- **strength**: fuerza del lazo afectivo entre los actores.

- **startTime**: momento en que se inicia la relación.

- **emotions**: instancia de la clase EmotionalState, que indica las emociones que suscita el actor destino en el actor origen.

ModifyRelationship: representa la modificación explícita (según la clase) de una relación entre dos actores. Los atributos propios que componen esta clase son:

- **destination**: actor destino al que va dirigida la relación.

- **source**: actor origen de la relación.

- **strength**: fuerza del lazo afectivo entre los actores.

- **kind**: tipo de relación que tiene lugar entre los dos actores. Puede ser basada en edad (AgeBased), estatus social (StatusBased) o parentesco (BloodBased).

- **startTime**: momento en que se inicia la relación.

- **emotions**: emociones que suscita el actor destino en el actor origen.

NonVerbalCommunication: Especialización de la clase **ClientAction** que representa una acción de comunicación no verbal entre dos actores. Los atributos propios que componen esta clase son:

- **destination**: actor hacia el que se dirige la acción comunicativa (receptor). El origen de la comunicación es el atributo **actor** heredado de la clase **ClientAction**.

- **emotion_valence**: valencia de la emoción transmitida.
- **emotion_arousal**: activación de la emoción transmitida.

VerbalCommunication: representa una acción de comunicación verbal. Los atributos de esta clase son:

- **destination**: actor hacia el que se dirige la acción comunicativa (receptor).
- **subject**: tema de la comunicación. Puede ser un objeto, acción, idea o actor.
- **emotion_valence**: valencia de la emoción transmitida.
- **emotion_arousal**: activación de la emoción transmitida.
- **strength**: fuerza de la emoción transmitida.

Communication: esta clase representa la acción de utilizar un objeto de comunicación. Consta de sólo un atributo, que es:

- **object**: objeto de la comunicación.

MoveAction: Especialización de la clase **ClientAction** que representa el movimiento de un actor de una celda a otra. Los atributos propios que componen esta clase son:

- **x**: coordenada X de destino del actor.
- **y**: coordenada Y de destino del actor.

ChangeStage: Especialización de la clase **ClientAction** que representa la acción de teletransportarse a otro escenario. Los atributos propios que componen esta clase son:

- **stage**: escenario al cual se va a cambiar.
- **actor**: actor que va a cambiar de escenario.

Telecarrier: Especialización de la clase **ClientAction** que representa la acción de usar un teletransportador (el objeto que permite el cambio entre escenarios). El único atributo propio que compone esta clase es:

- **teleporter**: objeto de teletransporte que va a ser usado.

FindFreeCell: Especialización de la clase **ClientAction** que representa la acción de desplazamiento a una celda libre. El único atributo propio que compone esta clase es:

- **destinationCell**: celda de destino libre a la que se desplazará el actor.

PickUpAction: Especialización de la clase **ClientAction** que representa la acción de recoger un objeto del escenario. El único atributo propio que compone esta clase es:

- **object**: objeto a recoger del escenario.

PutDownAction: Especialización de la clase **ClientAction** que representa la acción de dejar un objeto en el escenario. Los atributos propios que componen esta clase son:

- **object**: objeto a dejar en el escenario.
- **xObject**: coordenada X de la celda donde dejar el objeto.
- **yObject**: coordenada Y de la celda donde dejar el objeto.

PickUpFromContainer: Especialización de la clase **ClientAction** que representa la acción de recoger un objeto de un objeto contenedor. Los atributos propios que componen esta clase son:

- **object**: objeto a recoger del escenario.
- **container**: instancia del contenedor.

PutDownInContainer: Especialización de la clase **ClientAction** que representa la acción de dejar un objeto en un objeto contenedor. Los atributos propios que componen esta clase son:

- **object**: objeto a dejar en el contenedor.
- **container**: instancia del contenedor donde se deja el objeto.

Play: Especialización de la clase **ClientAction** que representa una acción de juego del actor. El único atributo propio que compone esta clase es:

- **leisureObject**: objeto de entretenimiento a utilizar.

Read: Especialización de la clase **ClientAction** que representa una acción de lectura del actor. El único atributo propio que compone esta clase es:

- **readingObject**: objeto de lectura a utilizar.

Working: Especialización de la clase **ClientAction** que representa la acción de trabajar con un objeto. El único atributo propio que compone esta clase es:

- **workObject**: objeto de trabajo a utilizar.

Washed: Especialización de la clase **ClientAction** que representa una acción de higiene personal. El único atributo propio que compone esta clase es:

- **bathObject**: objeto de higiene personal a utilizar.

Resting: Especialización de la clase **ClientAction** que representa una acción de descanso del actor. El único atributo propio que compone esta clase es:

- **restObject**: objeto de descanso a utilizar.

Continue: Especialización de la clase **ClientAction** que indica que un actor continúa realizando la acción anterior.

Drink: Especialización de la clase **ClientAction** que representa la acción de beber un objeto de bebida que tenga cogido el actor. Los atributos propios que componen esta clase son:

- **drinkobject**: objeto de bebida que va a ser utilizado en la acción.

Eat: Especialización de la clase **ClientAction** que representa la acción de comer un alimento que tenga cogido el actor. Los atributos propios que componen esta clase son:

- **food**: objeto de comida.

BuyDrink: Especialización de la clase **ClientAction** que representa la acción de comprar un objeto de bebida en una tienda. Los atributos propios que componen esta clase son:

- **drinkobject**: objeto de bebida que va a ser comprado.
- **shop**: tienda donde se encuentra almacenado el objeto a comprar.

BuyFood: Especialización de la clase **ClientAction** que representa la acción de comprar un objeto de comida en una tienda. Los atributos propios que componen esta clase son:

- **food**: objeto de comida que va a ser comprado.
- **shop**: tienda donde se encuentra almacenado el objeto a comprar.

Exit: Especialización de la clase **ClientAction** que representa la acción de salida de un actor en el juego. El único atributo propio que compone esta clase es:

- **exitObject**: objeto de salida del juego.

Object: Especialización de la clase **Entity** que representa un objeto, como una entidad del universo AI-LIVE. Actúa como clase padre para el conjunto de clases que se enumeran en el siguiente bloque, que representan los distintos tipos de objetos existentes. Cuando el objeto se puede utilizar en alguna de las acciones definidas para satisfacer las necesidades básicas de los actores, los atributos propios de esta clase indican el valor en que cada uno de los drives del actor que utiliza dicho objeto debe aumentar, disminuir o permanecer igual (valor cero). Esto se debe a que la utilización de un objeto, a pesar de disminuir el valor de una necesidad básica, puede a su vez aumentar el de otra (por ejemplo, jugar con una pelota disminuye el aburrimiento pero aumenta el cansancio y suciedad, al ser una actividad física).

Los atributos propios de la clase son los siguientes:

- **value_hunger**: valor de hambre que aumenta o disminuye al utilizar este objeto.

- **value_thirst**: valor de sed que aumenta o disminuye con la utilización del objeto.

- **value_dirtness**: valor de suciedad que se modifica al utilizar el objeto.

- **value_boredom**: valor de aburrimiento que se modifica al utilizar el objeto.

- **value_tiredness**: valor de cansancio que aumenta o disminuye utilizando un objeto de este tipo.

- **value_solitude**: valor de soledad que se modifica utilizando el objeto.
- **type**: tipo de objeto.

ComunicationObject: Especialización de la clase **Object** que representa un objeto de comunicación.

Phone: Especialización de la clase **ComunicationObject.** Representa un teléfono para poder realizar la acción de comunicación y ser representado gráficamente como un teléfono.

BathObject: Especialización de la clase **Object** que representa un objeto de aseo personal. Los atributos propios que componen esta clase son:

- **ocuppied**: indica si el objeto está ocupado (yes) o no (no).

- **actor**: actor que está ocupando el objeto. En caso de no encontrarse ningún actor dentro, el valor del atributo será nil.

Shower: Especialización de la clase **BathObject** que representa una ducha.

Bathub: Especialización de la clase **BathObject** que representa una bañera.

Washbowl: Especialización de la clase **BathObject**. Representa un lavabo.

Toilet: Especialización de la clase **BathObject.** Representa un retrete.

RestObject: clase que representa un objeto de tipo descanso, donde el actor descansará para disminuir el cansancio. Sus atributos son:

- **ocuppied**: indica si el objeto está ocupado (yes) o no (no).

- **actor**: instancia del actor que está ocupando el objeto. En caso de estar libre, el valor del atributo será nil.

Bed: Especialización de la clase **RestObject.** Representa una cama utilizada en las acciones de descanso para que el actor duerma.

Sofa: Especialización de la clase **RestObject.** Representa un sofá en el cual un actor puede descansar echándose una siesta. Disminuye el drive de cansancio en menor cantidad que durmiendo en una cama.

Food: Especialización de la clase **Object**, que representa un objeto de tipo comida, que el actor utilizará para disminuir su valor de hambre. Sus atributos son:

- **price**: precio del objeto de comida en caso de tener que comprarlo en la tienda.

Meat: Especialización de la clase **Food** que representa un objeto de comida de carne.

Fruit: Especialización de la clase **Food** que representa una fruta.

WorkObject: Especialización de la clase **Object** que representa un objeto para que el actor trabaje. De momento para trabajar se utiliza un ordenador. Sus atributos son:

- **value_wealth**: valor de riqueza o dinero que aumenta utilizando un objeto.

Computer: Especialización de la clase **WorkObject.** Representa un ordenador para que el actor pueda trabajar en él.

LeisureObject: Representa un objeto de tipo diversión, para que el actor disminuya el aburrimiento utilizando estos objetos.

Guitar: Especialización de la clase **LeisureObject.** Representa una guitarra para que el actor toque con ella, disminuyendo su aburrimiento.

Psp: Especialización de la clase **LeisureObject.** Objeto de ocio que representa una PSP para que el actor juegue con ella.

Ball: Especialización de la clase **LeisureObject.** Sirve para representar una pelota gráficamente, y que el actor juegue con ella.

Ipod: Especialización de la clase **LeisureObject.** Objeto que representa un Ipod para escuchar música, disminuyendo el aburrimiento del actor.

Chess: Especialización de la clase **LeisureObject**. Representa un tablero de ajedrez.

Dvd: Especialización de la clase **LeisureObject.** Representa un DVD.

Tv: Especialización de la clase **LeisureObject**. Representa una televisión.

Container: Especialización de la clase **Object** que sirve para representar a aquellos objetos que tienen la propiedad de contener a otros objetos. Los atributos propios que componen esta clase son:

- **objects**: lista de objetos contenida por el contenedor.
- **typeObjects**: lista de tipos de objetos (el literal, no una instancia) posibles que puede contener el contenedor. Por ej: Food, Meat, Water.
- **usedVolume**: representa el volumen usado dentro del contenedor.
- **maxLoad**: representa el número máximo de objetos que puede tener el contenedor.
- **usedLoad**: representa el número de objetos que almacena el contenedor.

Shop: Especialización de la clase **Container** que representa una tienda que contiene objetos de comida o bebida para que el actor los compre y consuma. El atributo *accessCell* heredado de la clase **Entity**, se utiliza para que el actor pueda entrar en la tienda.

Refrigerator: Especialización de la clase **Container** que representa un frigorífico que contiene objetos de comida. Utiliza el atributo accessCell para que los actores puedan acceder a él.

ReadingObject: Especialización de la clase **Object** que representa un objeto de tipo lectura.

Magazine: Especialización de la clase **ReadingObject**. Representa una revista.

Book: Especialización de la clase **ReadingObject**. Representa un libro.

DrinkObject: Especialización de la clase **Object** que representa un objeto de bebida. Los atributos propios que componen esta clase son:

- **price**: precio del objeto de bebida, utilizado para su compra en una tienda de bebidas.

Water: Especialización de la clase **DrinkObject** que representa una botella de agua. Sin atributos propios.

Wine: Especialización de la clase **DrinkObject** que representa un vaso de vino.

Coffee: Especialización de la clase **DrinkObject**. Representa una taza de café.

MiscellaneousObject: Clase padre que sirve para representar objetos que no se enmarcan en las clases anteriores. La mayoría de las especializaciones de esta clase no tienen una finalidad propia definida en el juego y sólo se utilizan en el interfaz gráfico como elementos decorativos o auxiliares.

Night_table: Especialización de la clase **MiscellaneousObject**. Representa una mesita de noche.

Washer: Especialización de la clase **MiscellaneousObject**. Representa una lavadora.

Microwave: Especialización de la clase **MiscellaneousObject**. Representa un microondas.

Kitchen: Especialización de la clase **MiscellaneousObject.** Representa una cocina.

Clock: Especialización de la clase **MiscellaneousObject**. Representa un reloj de pared.

Dresser: Especialización de la clase **MiscellaneousObject**. Representa un aparador.

4.4 Descripción del cliente CLIPS

Un cliente es una aplicación que accede a un servicio remoto, conocido como servidor, a través de una red de comunicaciones[7].

En AI-LIVE hay cuatro tipos de clientes que hemos denominado: clientes CLIPS, clientes Prodigy, clientes manuales y cliente GUI. Excepto el cliente GUI, que se encarga de representar gráficamente el desarrollo del juego, el resto de clientes modelan el comportamiento de un actor que de forma autónoma va realizando acciones en un escenario. La técnica utilizada para implementar el proceso de toma de decisiones de los actores determina los tipos de clientes considerados: los clientes CLIPS utilizan un sistema de producción, los clientes Prodigy utilizan técnicas de planificación automática y en los clientes manuales es el usuario el que decide las acciones a ejecutar.

Un sistema de producción consta de tres componentes: la base de hechos, la base de reglas y el motor de inferencia. Para los clientes CLIPS, la base de hechos está compuesta por las instancias de la ontología que representan el estado del universo

[7] Sadoski, Darleen. *Client/Server Software Architectures--An Overview*, Software Technology Roadmap, 1997-08-02

del juego. En cada turno del cliente CLIPS, el servidor le envía esta información a través de un mensaje con la etiqueta **STAT**.

Las acciones elegidas por el actor vienen determinados por las reglas de la base de reglas. Estas reglas se activarán o no según cumplan una serie de precondiciones necesarias. Las precondiciones, a su vez, vienen dadas por la existencia de hechos en el universo del juego, o bien por determinados parámetros de los

Las acciones que realizará el actor lanzado por el cliente vienen determinados por las reglas. Estas reglas se activarán o no según cumplan una serie de precondiciones necesarias. Las precondiciones, a su vez, vienen dadas por la existencia de hechos en el universo del juego, o bien de determinados parámetros de los atributos de los actores. El cliente no realiza ninguna modificación sobre ningún atributo o elemento del universo del juego, esto es realizado por el servidor.

Una vez ejecutada una regla, se almacenará la acción correspondiente a la regla en un fichero dinámico llamado *current.action*. Este fichero se envía al servidor a través de un mensaje con la etiqueta **ACTN**. El servidor al recibir la acción actualizará el estado del universo con la acción solicitada por el cliente. Para que el servidor pueda realizar las modificaciones en el estado, cada acción implementada en el cliente debe estar vinculada a una o varias reglas en el servidor.

Finalmente, cuando el cliente se desconecta, se genera un fichero de traza (*id_actor-TRAZA.txt)*, que se ha ido actualizando con la secuencia de acciones que ha realizado en cada turno.

Los elementos necesarios para que funcione un cliente CLIPS son:

- Una ontología, que recoge todas las clases del universo AI-LIVE y las relaciones entre ellas (*ontology.clp*). La ontología corresponde al apartado 4.3 ya descrito.

- El código fuente del cliente en CLIPS, donde se almacenan las reglas que permiten al cliente tomar decisiones por sí solo (*client.clp*).

- Un fichero con extensión *.profile* (por ejemplo, "*Amy.profile"*), que contiene datos sobre los atributos de una instancia de la clase AddClient, para que al conectarse al servidor se cree una instancia de la clase Actor.

- Un escenario al que conectarse. Es un fichero que contiene todas las instancias de los objetos y entidades que están en el escenario, información de las celdas en las que se sitúa, etc.

Un perfil es aquel fichero en el que se encuentra una instancia de la clase AddClient, comentada en el apartado anterior. La instancia almacena distintos atributos pertenecientes a un actor con unos valores prefijados. De esta manera se

permite crear una configuración inicial de los valores de los atributos del actor, y crear tantos ficheros de perfiles como se desee. Se encuentran en el directorio “profiles” del proyecto, cuya extensión de los ficheros que contiene se denomina *.profile*. A continuación mostramos el contenido de uno de los ficheros:

```
([DEFAULT_ACTOR] of AddClient
(name_ "Amy")
(gender F)
(entityGraphic "casual woman")
(maxLoad 5)
(maxVolume 10)
(volume 1)
(weight 55)
(agreeableness 1.0)
(conscientousness 1.0)
(extraversion 1.0)
(neuroticism 0.0)
(openness 1.0)
(age 0)
(status 0)
(sex 0)
(arousal 0.0)
(valence 0.0)
(dislikenessList Book Meat Bathtub [Working])
(likenessList Shower Bed Water [Resting])
(neutrallikeList Ball)
(inter_likeList Fruit)
(inter_dislikeList Sofa [Washed])
(maxhunger 30)
(maxthirst 30)
(maxdirtness 30)
(minwealth 30)
(maxtiredness 30)
(maxboredom 30)
(maxsolitude 30)
(continueProbability 80)
```

4.5 Interactividad con objetos del escenario

En el estado anterior del proyecto, los actores eran dueños de una bolsa personal que se les asignaba automáticamente al conectarse con el servidor y llevaban encima durante todo el juego. Esta bolsa contenía ciertos objetos, que se correspondían con objetos de comida, de entretenimiento o de lectura; de tal forma que cuando quisieran utilizarlos sólo tenía que coger el objeto de la bolsa sin ser necesario que se moviera. Los únicos objetos que se podían acabar de la bolsa eran los de comida, que tras ser consumidos por el actor, los debía comprar en la tienda del juego.

Para ofrecer una mayor riqueza en la toma de decisiones de los actores de AI-LIVE, se optó por eliminar la bolsa. Que el actor lleve una bolsa encima con objetos no es necesario teniendo como universo del juego un escenario con objetos disponibles

repartidos en él. Eliminar la bolsa supone una mayor interacción con los objetos del escenario, hay más objetos para compartir y ningún objeto tiene dueño específico.

Además, entre los atributos del actor existían algunos que nunca se les había dado ninguna utilidad, por lo que resultaba de interés utilizarlos. Estos atributos son:

- **catchedObjects**: es la lista de objetos que ha cogido el actor y que lleva encima en ese momento. La bolsa tenía una clase propia en la que uno de sus atributos era una lista de objetos que estaban dentro.

- **usedLoad**: indica el peso total actual de los objetos que tiene cogidos el actor.

- **maxLoad**: indica el peso máximo de objetos capaz de cargar el actor.

Con la eliminación de la bolsa y el uso de estos atributos se pretende que para que un actor utilice un objeto que necesita, tenga que localizar primero la celda de acceso en la que está el objeto (o contenedor que contenga a dicho objeto), y después moverse a dicha celda de acceso. En caso de que el peso del objeto a coger, sumado al peso de los objetos que el actor ya lleva, supere el peso máximo del conjunto de objetos que puede llevar (definido en su atributo), le obligaría a dejar uno o más objetos en el suelo o en su respectivo contenedor.

También se crearon nuevas clases o reglas, o se modificaron ciertos aspectos de la ontología o estados. Todos estos cambios están relacionados con el fin de que el actor interactúe con los objetos y que pueda cogerlos y soltarlos. A continuación enumeramos estos cambios.

- Se eliminó de la **ontología** la clase correspondiente a la bolsa, así como los atributos de las clases en las que aparecía (por ejemplo, el atributo de la bolsa que estaba asignada al actor, en la clase Actor).

- **Regla** `pickUpAction`: El cliente, gracias a esta regla, puede decidir coger un objeto del escenario para usarlo o llevarlo encima. Para que la regla se active es necesario que exista una instancia de la acción *PickUpAction*; un escenario con entidades en el que una de ellas sea el objeto a coger; un actor en ese escenario; y un objeto que esté en la misma celda en la que se encuentra el actor actualmente (previamente el actor para coger un objeto ha tenido que desplazarse a la celda en la que está dicho objeto). Posteriormente se comprueba que el peso del objeto, sumado al peso total de objetos que lleva el actor (*usedLoad*), no supere el valor máximo permitido (*maxLoad*).

En caso de haberse activado la regla, se imprimirá que el actor ha cogido el objeto y se modificará la instancia de la acción, que será enviada al servidor para realizar la acción y modificar el estado del juego.

Esta misma regla en el lado del servidor (desarrollado paralelamente a este proyecto), actualizará el estado del escenario y del actor, añadiendo el objeto a la lista de objetos cogidos por el actor (*catchedObjects*), y actualizando el valor de *usedLoad*.

El objeto se eliminará de la lista de entidades del escenario, ya que ahora está en propiedad del actor.

- **Regla** `putDownAction`: esta acción permite al actor dejar un objeto en el suelo, bien porque no lo necesite o porque sea necesario dejarlo para poder coger otro, debido a que no pueda soportar más peso. Las precondiciones para que se active la regla son que exista una instancia de la acción, un escenario en el que esté el actor, una celda adyacente al actor que esté desocupada, y el objeto a dejar en el suelo. Se comprueba que el objeto está en posesión del actor. Si se activa la regla, en el consecuente se imprime información sobre la acción realizada y se modifica el hecho correspondiente a la acción para pasar la información al servidor.

El servidor se encargará de actualizar el estado del actor y del escenario, eliminando el objeto a dejar de la lista de objetos cogidos por el actor, y restando el peso del objeto al total que lleva cargado el actor. Además, el objeto vuelve a ser insertado en la lista de instancias del escenario, indicando de esta manera que está libre en una celda para ser cogido.

- **Regla** `pickUpFromContainer`: La implementación de esta regla estuvo motivada por la necesidad de que el actor pudiera coger objetos de un contenedor para llevarlo encima y usarlo cuando lo necesitara.

Inicialmente tras la eliminación de la bolsa, se optó por implementar una regla llamada *PickUpFood* (igual para los objetos de bebida con *PickUpDrink*), la cual tras localizar un objeto de comida, el actor cogía un objeto de comida del frigorífico y lo consumía en el mismo turno. Esta situación no se ajustaba a la realidad, ya que el actor puede o debe permitir coger comida del frigorífico y guardarlo para comérselo posteriormente. Se decidió por tanto implementar una regla genérica para coger cualquier objeto que se encuentre en un contenedor para cogerlo.

Las diferencias respecto a la regla *PickUpAction* residen en varias precondiciones, siendo en este caso necesario que el objeto localizado previamente por el actor se encuentre en un contenedor, el cual está en el mismo escenario que el actor. Obviamente debe cumplirse también que el peso del objeto seleccionado no supere el peso máximo de los objetos que puede soportar el actor. Una vez activada la regla y ejecutada ésta, se envía al servidor una instancia de la acción *PickUpFromContainer* para que actualice el estado del juego asignando el objeto a coger a la lista de objetos cogidos por el actor.

- **Regla** `putDownInContainer`: Al igual que la regla anterior, se implementó también la acción de depositar objetos en un contenedor.

La particularidad de esta regla se debe a que se ha decidido que el actor deba dejar un objeto en su correspondiente contenedor. Es decir, la lógica dice que los objetos de comida o bebida deben almacenarse en la nevera o algún armario dependiendo del tipo; no se debe permitir que el actor pueda depositar, por ejemplo, un objeto de ocio como una pelota o una guitarra en este tipo de contenedores.

Esta precondición se cumple cuando el tipo del objeto a dejar, pertenece a uno de los tipos objetos permitidos del objeto contenedor (se trata de una lista con los tipos de objetos, definido en la ontología). El resto de precondiciones son que el actor esté en la celda de acceso del contenedor, y que tenga un objeto cogido, el cual es el que se depositará.

- **Regla** `putDownGoal`: Esta regla sirve para localizar un contenedor en el cual se pueda depositar uno de los objetos que tiene cogidos el actor.

Su funcionamiento es parecido al de la regla anterior, donde las precondiciones son que el actor tenga cogido un objeto cuyo tipo sea el mismo que uno de los tipos permitidos de algún contenedor del escenario.

En el consecuente de la regla se actualiza la instancia de la clase Target, con los datos de la celda de acceso y objeto a dejar, para tener la información completa y que el actor se mueva en la siguiente acción.

- **Clase** `pickUpAction`: Como se ha mencionado en la regla *pickUpAction*, una de las precondiciones para la activación de la regla es que existiera una instancia de la acción. Esta instancia se debe a la clase implementada **pickUpAction**. Es una especialización de la clase **ClientAction**, por lo que hereda sus atributos. Como atributo propio tiene el siguiente:

  - **object**: este atributo indica el objeto que el actor va a coger. Será una instancia de la clase **Object**.

- **Clase** `putDownAction`: Al igual que la clase anterior, es necesario implementar una clase para que exista una instancia de este tipo y que se active la regla *putDownAction*. Por tanto, la clase **putDownAction** es también una especialización de la clase **ClientAction**, cuyos atributos propios son:

  - **object**: este atributo indica el objeto que el actor dejará en el escenario. Será una instancia de la clase **Object**.
  - **xObject**: atributo para indicar la coordenada x del escenario donde dejar el objeto.
  - **yObject**: atributo para indicar la coordenada y del escenario donde dejar el objeto.

- **Clase** `pickUpFromContainer`: Esta clase contiene los atributos necesarios para recoger la información de la acción *PickUpFromContainer*, que el cliente envía al servidor. A continuación se describen sus atributos propios, ya que cuenta con otros heredados de ClientAction.

  - **object:** este atributo indica el objeto que el actor va a coger. Será una instancia de la clase Object.
  - **container**: instancia del contenedor que contiene el objeto que se va a coger.

- **Clase** `putDownInContainer`: Definición de la clase para recoger una instancia de una acción *putDownInContainer*. Esta acción tiene la particularidad con respecto a la acción de coger objetos *PickUpAction*, que se utiliza un contenedor. Por tanto, definimos a continuación sus atributos propios:

  ~ **object**: este atributo indica el objeto que el actor va a depositar. Será una instancia de la clase **Object**. El objeto debe coincidir en tipo con los distintos tipos de objetos que puede almacenar el contenedor.

  ~ **container**: instancia del objeto contenedor en el que se va a depositar el objeto de la acción.

- **Clase** `Container`: Se ha modificado esta clase para añadir un nuevo atributo. Dicho atributo sirve para controlar que un actor deje un objeto en su correspondiente contenedor, como se ha comentado previamente en la regla *putDownInContainer*. Se describe de la siguiente forma este nuevo atributo:

  ~ **typeObjects**: Multislot donde cada elemento es una cadena de caracteres que indica el tipo de objeto que puede contener.

- **Regla** `Play`: Esta regla sufrió varias modificaciones para que su activación sea efectiva. A las precondiciones existentes anteriormente se ha añadido la precondición de que el objeto de juego esté en posesión del actor (figure una instancia de un objeto de ocio en la lista *catchedObjects* del actor), eliminándose por tanto la precondición de que el objeto estuviera en la bolsa del actor.

- **Regla** `Read`: Se modificó esta regla al igual que la anterior, eliminando la precondición de que el objeto de tipo ReadingObject debía encontrarse en la bolsa, y añadiendo la precondición de que la instancia del objeto se encuentre en la lista de objetos cogidos por el actor.

- **Método** `printPickUpFromContainer()`: este método sirve para imprimir en el fichero de traza los valores de los atributos de una instancia de la clase **PickUpFromContainer**, de los que se imprime: el escenario donde se encuentran actor y contenedor, el ID del actor, el objeto que va a coger, y el contenedor que contiene dicho objeto. Es llamado desde el consecuente de la regla *PickUpFromContainer*, para reflejar la acción realizada en la traza del fichero.

- **Método** `printPutDownInContainer()`: este es el método al que se llama desde la regla *PutDownInContainer*. Es el encargado de imprimir en fichero los datos significativos de la acción realizada, reflejados en una instancia de la clase **PutDownInContainer**. Se corresponden con los mismos atributos de la clase anterior,

donde el objeto en cuestión se corresponde con el objeto que se va a dejar en un determinado contenedor.

- Por último, se modificó el estado inicial del escenario, eliminando las instancias existentes de las bolsas (ya que se eliminó esa clase), y se recolocaron los objetos que se incluían en las bolsas en otras celdas del escenario (principalmente los objetos de lectura y ocio). Los objetos de comida se incluyeron dentro del frigorífico (objeto contenedor).

4.6 Drive de necesidad básica para la sed del actor

Se ha añadido otro drive de necesidad básica para los actores, para permitir el uso de más objetos. Se trata del drive **thirst** (sed).

Para satisfacer la sed del actor y completar la funcionalidad de este nuevo drive se han creado las siguientes clases y reglas que se enumeran a continuación, comenzando por las clases referentes a los objetos existentes, y siguiendo con las clases y reglas propias de las acciones que puede realizar el actor:

- **Clase** `DrinkObject`: Es una especialización de la clase **Object**, y a su vez la clase padre de los objetos de tipo bebida del sistema. Sus atributos propios son:

  ~ **price**: valor del objeto en la tienda cuando el actor acude a comprarlo.

Gracias a los atributos heredados de Object, permiten a las instancias de objetos de bebida que se creen posteriormente añadir la información sobre el valor de sed, hambre, etc, que afecta en el actor cuando usa dichos objetos.

- **Clase** `Wine`: Clase que representa uno de los posibles tipos de bebidas del sistema, en este caso vino. Es por tanto, una especialización de la clase **DrinkObject**. No consta de atributos propios.

- **Clase** `Water`: Clase que representa un vaso de agua. Es una especialización de la clase **DrinkObject**. No tiene atributos propios.

- **Clase** `Coffee`: Clase que representa una taza de café. Es una especialización de la clase **DrinkObject**. No tiene atributos propios.

- **Clase** `Drink`: Representa la acción del actor de beber una bebida. Una instancia de esta clase sirve, por tanto, para que el cliente comunique al servidor que

ha realizado esta acción. Es una especialización de la clase **ClientAction** y tiene como atributos propios:

- **drinkobject**: objeto de bebida que bebe el actor.

- **Clase** `BuyDrink`: Clase que representa la acción de comprar una bebida en una de las tiendas del juego. Al igual que la anterior clase, al representar una acción es una especialización de la clase **ClientAction**. Sus atributos propios son:

- **drinkobject**: objeto de bebida que compra el actor. El objeto pasará a la lista de objetos cogidos por el actor (de esto se encarga de actualizarlo el servidor).
- **shop:** instancia de la tienda donde comprar objetos.

- **Regla** `DrinkObjectGoal`: Mediante esta regla, el actor establece como objetivo coger un objeto de bebida y posteriormente se moverá a la celda en la que se encuentre dicho objeto, o celda adyacente si el objeto se almacena dentro de un objeto contenedor. La principal precondición para que se active la regla consiste en que el valor del drive *thirst* del actor debe ser mayor que el umbral del mismo drive, que tiene definido en su perfil (visto anteriormente en el fichero *.profile* del actor). Este umbral, junto con el resto de umbrales de los drives, se encuentra en una instancia de la clase **ActorCounters**, cuyo tipo es *Continue*. El resto de precondiciones son las siguientes:

- una instancia de la acción MoveAction.
- Una instancia de la clase Target (clase auxiliar creada en el código CLIPS del cliente), que contiene un objetivo (objeto), una celda objetivo, y un atributo para activar o no esta instancia.
- Un frigorífico en el escenario.
- Un objeto de bebida que debe pertenecer a la lista de objetos del frigorífico.
- Una celda adyacente al frigorífico que esté desocupada, para que el actor se mueva hasta ella.

En el consecuente de la regla se modifica la instancia de la clase Target para que la siguiente regla en activarse sea la que trata de la acción de moverse a su objetivo final, en este caso dirigirse al frigorífico a por una bebida.

- **Regla** `DrinkShopObjectGoal`: al igual que la regla anterior, esta regla sirve para marcar como objetivo del actor comprar una bebida en la tienda para saciar su necesidad de sed o utilizarla más tarde. Esta regla se activará cuando se hayan acabado en el frigorífico las bebidas, ya que por la prioridad de las reglas (aunque es algo configurable), primero acudirá a coger bebida del frigorífico mientras haya. Las precondiciones para la activación de la regla son:

- una instancia de la acción MoveAction.
- Una instancia de la clase Target.

- Una tienda en el escenario, el mismo en el que se tiene que encontrar el actor.
- Un objeto de bebida que debe pertenecer a la lista de objetos de la tienda.
- Una celda adyacente a la celda, que esté desocupada.
- El actor debe tener un valor de sed actual mayor que el del atributo *thirst* de la instancia donde se recogen los valores de los umbrales de los drives, para indicar que quiere satisfacer su necesidad;
- Debe tener una cantidad de dinero mayor o igual al precio del objeto de bebida.

El consecuente de la regla imprime la acción a realizar, indicando la situación actual del actor, y la celda de la tienda y del objeto de bebida a comprar, siendo allí donde se moverá posteriormente. Finalmente se modifica la instancia de la clase Target para actualizar los atributos y ser después utilizada en la regla que implica al actor moverse a su objetivo.

- **Regla** `Drink`: esta es la regla en CLIPS que permite que el actor beba un objeto de bebida para disminuir su sed. Las precondiciones necesarias para que la regla se active son:

- una instancia de la acción.
- Un escenario en el cual tiene que estar el actor.
- Un objeto de bebida que debe pertenecer a la lista de objetos cogidos por el actor. Para ello anteriormente el actor ha cogido el objeto mediante la regla *PickUpAction* o *PickUpFromContainer*.
- El drive de sed del actor debe ser igual o mayor al valor del umbral del drive definido en su perfil (que se recoge en una instancia de la clase ActorCounters).

El consecuente de la regla imprime los datos de la instancia de la acción realizada, y se modifica dicha instancia para comunicarle al servidor la acción realizada.

- **Regla** `buyDrink`: mediante esta regla el actor compra un objeto de bebida de una tienda existente en el juego. Independientemente de la cantidad de objetos de bebida que queden en el frigorífico, puede acudir a la tienda a comprar, bien para consumir la bebida después o tenerla cogida para posteriormente usarla. Las precondiciones de la regla son:

- una instancia de la acción.
- Un escenario en el cual tiene que estar el actor.
- Una tienda, que debe estar también en el mismo escenario.
- El actor debe estar situado en la celda de acceso a la tienda.
- Un objeto de bebida que debe pertenecer a la lista de objetos cogidos por el actor. Para ello anteriormente el actor ha cogido el objeto mediante la regla *PickUpAction* o *PickUpFromContainer*.

- El drive de sed del actor debe ser igual o mayor al valor del umbral del drive definido en su perfil (que se recoge en una instancia de la clase ActorCounters).
- Además, el actor debe tener tanto dinero o más del precio del objeto para poder comprarlo.

En el consecuente de la regla, se modifica la instancia de la acción con el objeto comprado para enviárselo al servidor y que actualice el estado. También se imprimen los datos de la instancia de la acción para guardarlos en la traza.

- **Método** `printDrinkAction()`: este método abre el fichero de la acción actual (*current.action*) en modo escritura, y con una instancia existente de la clase **Drink** imprime en el fichero cada atributo de la instancia: escenario en el que se encuentra, el actor que realiza la acción, y la bebida consumida por el actor. Es llamado desde el consecuente de la regla *Drink*.

- **Método** `printBuyDrinkAction()`: este es el método al que se llama desde la regla *buyDrink*. Es el encargado de imprimir en fichero los valores de los atributos de una instancia de la clase *BuyDrink*, que son: el escenario donde están tienda y actor, el actor que compra la bebida, la bebida comprada y tienda que tenía la bebida.

4.7 Incorporación del modelo comunicativo emocional. Drive "solitude"

Al incorporar el modelo comunicativo emocional [Jimenez, 2008] a nuestro sistema, se observó que el funcionamiento no era el adecuado. Este modelo funcionaba siempre con un mínimo de dos clientes en el escenario para permitir la comunicación entre ellos, ya que el modelo estaba destinado a ese fin. Esto implicaba que sería necesario lanzar un mínimo de dos clientes en una ejecución para poder utilizar acciones de comunicación. En ese caso, además, sólo se activaban las acciones de comunicación entre actores, sin que se pudieran activar el resto de acciones que cubren la satisfacción de las necesidades básicas.

El problema surgido, y por tanto el objetivo a alcanzar era encontrar la manera de hacer compatibles los dos modelos para dotar de mayor riqueza a la aplicación.

En primer lugar se llegó a la conclusión de que en el estado actual, a un actor no le resulta indispensable comunicarse con otro, ya que se trata de acciones que no requieren precondiciones (un cierto nivel de hambre, cansancio, etc); por lo que el resto de acciones para cubrir las necesidades básicas se activarían antes (en caso de tener alguna necesidad básica insatisfecha). Una de las soluciones, era la de "obligar" al actor, a comunicarse con otros agentes.

Para ello, se ha creado un nuevo drive de necesidad básica para el actor: **solitude**. Solitude representa la soledad del actor, o dicho de otra forma, la falta de relación social con otros actores del escenario. Para satisfacer esta necesidad será necesario comunicarse con otros actores, lo que dará importancia a las acciones del modelo comunicativo emocional.

Al crear este drive también se encontró con otro problema: al lanzar un solo cliente en una ejecución, éste no tenía oportunidad de realizar acciones de comunicación con otros actores, ya que no habían más en el escenario (hasta que otro cliente se conectara). Esto implicaba que el nivel de soledad iría siempre en aumento, sin poder satisfacer la necesidad de ninguna manera; lo que influye a la hora de realizar otras acciones, en las cuales el nivel de soledad puede ser una de las precondiciones necesarias para realizar dicha acción.

Una de las soluciones a este problema, fue la de añadir un objeto que pueda cumplir la funcionalidad de comunicación, que pueda disminuir el nivel de soledad del actor. Se determinó, por tanto, utilizar un teléfono, gracias al cual el actor puede hablar para relacionarse socialmente. Para ello, era necesario crear un tipo de objeto nuevo, implementando las siguientes clases en la ontología y reglas en el código del cliente:

Clase `CommunicationObject`: Especialización de la clase **Object** que servirá para representar un objeto destinado a la comunicación. No consta de atributos propios, siendo los heredados de su clase padre los que utiliza. Entre ellos el más importante es el que indica el valor de soledad que disminuye en el actor al utilizar el objeto (value_solitude).

Clase `Phone`: Es el objeto que representa un teléfono para que un actor pueda comunicarse. Es una especialización de la clase **CommunicationObject** y no contiene atributos propios.

Es posible ampliar la ontología con otros objetos que permitan comunicarse a un actor. Para ello habría que crear clases de objetos que sean especializaciones de la clase **CommunicationObject**.

Para realizar la acción comunicativa utilizando estos nuevos objetos, se ha implementado también la clase **Communication**, que representa dicha acción. Por lo tanto, y como cualquier acción que puede realizar un cliente, es una especialización de la clase **ClientAction**. Como atributos propios contiene:

- **object**: es la instancia del objeto de tipo CommunicationObject que se utiliza para la comunicación.

Las reglas que se implementaron para la utilización de estos objetos fueron las siguientes:

Regla `CommunicationObjectGoal`: Regla en la cual el cliente localiza un objeto que permita comunicarse con otros actores, para moverse hacia dicho objeto posteriormente y cogerlo en el siguiente turno. Para ello, se debe cumplir:

- Que existe una instancia de tipo **MoveAction**, y una celda de destino para moverse hasta dicha celda.
- También debe existir un actor, que se encuentre en el escenario.
- Existe al menos un objeto de tipo **CommunicationObject** en el escenario, en una celda de acceso que esté libre.
- El drive de soledad del actor debe ser igual o mayor al valor del umbral del drive definido en su perfil (que se recoge en una instancia de la clase **ActorCounters**).

El consecuente de la regla imprimirá los datos de interés de la acción, como son el actor, el objeto y celdas de inicio y destino que participarán en la acción de moverse del actor. Se modificará también una instancia de la clase auxiliar **Target** para actualizar la celda de destino y el objeto a coger, y así la regla *Move* que desplaza al actor de una celda a otra, tenga la información completa de la instancia, la cual es una de sus precondiciones.

Regla `Communication`: Regla que trata la acción donde el cliente se comunica con otro actor utilizando un objeto de comunicación. Se debe de cumplir:

- que exista una instancia de tipo Communication.
- un actor que haya cogido un objeto de tipo CommunicationObject y que por tanto lleve encima.
- y que el actor tenga un valor del drive de soledad suficiente, que haga que necesite comunicarse con otro actor.

En el consecuente de la regla se modifica la instancia de la clase **Communication** para enviar al servidor, y que actualice el estado con la acción. También se imprimen los atributos de la acción, recogidos en su instancia, para escribirlo en la traza. Esto se realiza a través de un método:

Método `printCommunicativeAction()`: Método que imprime los valores de los atributos de la instancia de la clase **Communication** que exista. Estos son: el escenario, el actor y el objeto de comunicación utilizado.

También se contempló que resultaba más lógico que los actores, a la hora de hablar entre ellos, estuvieran cerca el uno del otro. Anteriormente la precondición que existía para la activación de reglas que tratan de acciones comunicativas, era únicamente que debía existir en el escenario otro actor, por lo que una vez activada la regla, el actor se disponía a hablar con el otro, pudiendo estar cada uno en cualquier celda del escenario. Para solucionar esto, se implementó una nueva regla:

Regla `ActorGoal`: Esta regla hace que el actor localice a otro actor en el escenario, para acercarse a él, con el objetivo de hablar. Para ello debe activarse la necesidad de disminuir su soledad, así como debe de existir otro actor en el escenario, como precondiciones más importantes. El resto son:

- Debe existir una instancia de la acción **MoveAction**.
- También debe de existir una instancia de la clase **Target**.
- Ambos actores deben estar en el mismo escenario.
- Al menos una de las celdas adyacentes al actor destino debe de estar libre (sin objetos o entidades en ella).

El consecuente de la regla actualiza la instancia de la clase **Target**, actualizando los atributos de la celda de destino y el objetivo del actor, para que en la regla posterior el actor se mueva hasta la casilla destino.

4.8 Implementación del concepto del tiempo en el sistema

Uno de los conceptos a la hora de conseguir hacer de AI-LIVE un sistema más complejo era la implementación del tiempo. Nuestro sistema se basa en turnos, teniendo como idea de futuro implementar el sistema en tiempo real. Por tanto, la implementación del tiempo se presentaba en plantear un diseño para gestionar estos turnos. Realizar un sistema en el que las acciones duran distinto intervalo de tiempo dota al juego de mayor riqueza, ya que posibilita implementar más acciones, conseguir distintas ejecuciones y que el actor sea capaz de interactuar más con ellas. El objetivo es que por cada ejecución el actor tome distintas decisiones, influido por la distinta duración de las acciones.

A continuación se detallará el diseño planteado en conjunto por el cliente y el servidor, para entender en una visión global lo que se pretende con esta implementación, y posteriormente se detallará la parte que implica al cliente.

El diseño fue planteado de la siguiente manera, que describimos con una tabla y su posterior explicación (los valores son datos obtenidos de una ejecución):

Drive	*Valor mínimo drive*	*Valor máximo drive*	*Incremento drive por turno*	*Activación en cliente de la necesidad*
Hunger	0	100	1	60
Thirst	0	100	2	80
tiredness	0	100	1	80
dirtness	0	100	1	50
boredom	0	100	2	30
solitude	0	100	1	40

Figura 11: Ejemplo de organización de drives para la implementación del tiempo.

Drive	*Acción*	*Duración de acción*	*Valor disminución acción por turno*
Hunger	Eat	1	20
Thirst	drink	1	30
Tirdeness	Sleep	8	12
	Snap	4	20
Dirtness	Shower	3	30
Boredom	Play	2	10
Solitude	communication	3	15

Figura 12: Ejemplo de organización de turnos en acciones para la implementación del tiempo.

Cada fila de la tabla corresponde a un drive de necesidad básica. Este drive tendrá un valor mínimo, en el cual la necesidad está cubierta al completo, y un valor máximo, donde la necesidad necesita ser cubierta con la máxima prioridad. Estos valores mínimo y máximo están relacionados con una implementación realizada en el servidor. Dicha implementación se corresponde con la manera en la que afectan los drives en su conjunto al estado emocional del actor. Es decir, cuanto más cercano al valor mínimo estén todos los drives, afectará al estado emocional del actor de manera positiva, estando más alegre, etc., mientras que si los valores están cercanos al máximo, esto hará que el estado emocional sea negativo.

Para conseguir esta funcionalidad, el servidor calcula en cada turno un valor que implicará a la valencia y al arousal. Este valor, denominado *welfare* (bienestar) se obtiene mediante una fórmula en la que intervienen los valores de todos los drives. El parámetro *welfare* está ajustado para que el estado emocional no dependa totalmente de él, si no que afecte en una menor medida pero se haga notorio el cambio del estado emocional cuando el actor tiene satisfechas sus necesidades básicas o no.

Los drives tendrán un incremento constante en cada turno, independientemente de la acción realizada por el actor. De esta manera se simula la influencia del paso del tiempo en el actor, aumentando poco a poco su cansancio, hambre, suciedad, etc.

Ciertas acciones tendrán más duración que otras. Como se ha indicado anteriormente, la duración de todas inicialmente era de un turno. Esto permite acercarse al tiempo real que utilizamos cotidianamente para satisfacer nuestras necesidades básicas, ajustando de manera proporcional la duración de estas acciones a nuestro sistema, ya que no es lo mismo el tiempo empleado en descansar que en ducharse o comer, por ejemplo.

En la figura 12 se puede observar un ejemplo de las acciones y los turnos establecidos para cada acción. Las acciones que disminuyen un mismo drive, como por ejemplo snap y sleep para disminuir el cansancio, no tienen la misma duración, ya que no se emplea el mismo tiempo en dormir que en echarse una siesta; al igual que para

disminuir el drive de suciedad, ya que el tiempo en ducharse es menor al tiempo utilizado en bañarse en una bañera.

Por cada turno de la acción se decrementará un cierto valor al drive que esté relacionado con la acción, siendo la suma de cada turno el total que podrá disminuir el drive, ya que el actor podrá decidir no continuar con la acción y por tanto no disminuir el drive al máximo permitido por la acción. Además de esto, se ha optado que las acciones no impliquen sólo la disminución del valor de los drives, si no el aumento de otros. Por ejemplo, en acciones como jugar con la pelota, al actor le disminuirá su aburrimiento, pero aumentará el cansancio, suciedad o hambre, al suponer dicha acción un esfuerzo físico. Estos valores se definen en el estado inicial del juego, donde se encuentran las instancias de cada objeto del escenario con sus valores.

Finalmente, cada drive tendrá un valor de activación, a partir del cual el actor sentirá tener que cubrir la necesidad básica relacionada con el drive.

4.8.1 Nombres de los drives

En la versión anterior de AI-LIVE, los drives definidos no seguían criterios homogéneos respecto a su variación para conseguir la felicidad de los actores. Por ejemplo, mientras que el drive de hambre debía disminuir, los drives de energía, higiene y diversión debían subir. Se ideó que lo mejor es que los valores de todos los drives debían seguir el mismo criterio, y en particular se optó por la disminución. De esta forma, se cambió el drive de energía por el de cansancio, el de higiene por suciedad y el de diversión por aburrimiento y se añadieron los drives de sed y soledad.

En consecuencia, lo que se pretende indicar es que cuanto mayor es el valor del drive, mayor es la importancia de cubrir esa necesidad básica. Por ejemplo ya no tendría sentido decir que aumenta la diversión, cuando queremos decir que cuanto mayor sea el valor, más necesidad tiene el actor de divertirse. Se debía entonces modificar los nombres de los drives para que tuvieran sentido. Los drives que resultaron modificados fueron: diversión por aburrimiento (*boredom*), higiene por suciedad (*dirtness*) y energía por el cansancio (*tiredness*).

Algunos de los cambios realizados fue el de modificar los valores de los objetos que influyen en los drives, para su correcto funcionamiento con la nueva implementación.

Ya se dispone por tanto de los drives con los criterios unificados, con todos sus valores incrementándose. Si en futuras versiones se añadieran más drives, se deberá tener en cuenta estos aspectos.

4.8.2 Implementación del tiempo en las acciones

Nuestro sistema está basado en turnos que el servidor gestiona homogéneamente entre todos los clientes conectados. En cada turno, el actor correspondiente elige la acción que quiere realizar y el servidor simulaba su ejecución inmediata, es decir, todas las acciones duraban un turno. Para hacer funcionar el concepto del tiempo, uno de los pasos consistía en conseguir que determinadas acciones duren más de un turno, para asemejar la duración de estas acciones en el universo AI-LIVE a las tareas que realizamos cotidianamente. A continuación mencionamos las modificaciones realizadas.

Se implementó la clase **Continue**, para representar la continuación de la última acción elegida por el actor. Es una subclase de **ClientAction** sin atributos propios.

A la clase **Actor** se le añadieron los siguientes atributos:

- **continueProbability**: este atributo contiene un valor de probabilidad, que se puede definir previamente en el perfil del actor (fichero *profile*). De no definirse se genera automáticamente con un valor aleatorio cuando el cliente se conecta, pero ello implica tener que editar el código del servidor si se desea cambiar. Por tanto se recomienda utilizar la primera opción, ya que es configurable por el usuario.

  Se utiliza en la regla **Continue** (que detallaremos después), y permite al actor abandonar lo que estaba haciendo y realizar una acción distinta, sin que tenga que utilizar todos los turnos de la acción. Cuanto mayor sea el valor de *continueProbability*, mayor es la probabilidad de que el actor realice la acción todos los turnos que dura.

- **turn**: Este atributo sirve para llevar el número de turnos consumidos respecto a la última acción realizada. Por ejemplo, si una acción dura cuatro turnos, cuando realice esa acción el contador se pondrá a tres, decrementándose en los turnos sucesivos del actor hasta llegar a cero. Si la acción se interrumpe el contador se actualizará a cero y se inicializará nuevamente cuando realice otra acción cuya duración sea mayor a un turno.

Regla `continue`: Esta regla permite continuar con la acción anterior, en caso de que dicha acción dure más de un turno. Para que se pueda activar, las precondiciones son:

- Que el actor envíe una acción de tipo Continue.
- Que exista un escenario donde esté el actor y la acción.

- El atributo *turn* del actor debe ser mayor que cero, lo que indica que es una acción que dura más de un turno.
- Se crea un valor aleatorio con la función *random*, que es comparado con el valor del atributo *continueProbability*. La condición es que el valor aleatorio generado sea menor o igual que el valor de *continueProbability.*

El consecuente de la regla imprime la acción que ha realizado el actor, y modifica los atributos de la instancia de la acción para enviársela al servidor, quien es el encargado de restar el valor de *turn* y de actualizar los drives del actor.

4.8.3 Activación de necesidades en los actores

Se ha decidido a la hora de implementar el cliente, utilizar como estrategia en la toma de decisiones del actor para cubrir sus necesidades básicas, el uso de variables que tengan unos valores predefinidos que permitan la activación de las necesidades básicas del actor. Las reglas para satisfacer las necesidades básicas de los actores sólo se dispararán cuando el drive correspondiente tome un valor mayor o igual al de activación. Anteriormente, estos valores se definían en las propias reglas en el código CLIPS del cliente. Esto presentaba el inconveniente de que si se quería modificar el valor de activación, se debía de modificar manualmente en el fichero del código CLIPS del cliente, en la regla deseada.

Por tanto, una de las modificaciones realizadas al respecto ha sido la de crear nuevos atributos para el actor, que sirven para configurar y prefijar el nivel de las activaciones de estas necesidades. Eso presenta como ventaja que el usuario únicamente debe de acceder al fichero del perfil correspondiente al actor que quiera lanzar, y modificar estos valores. Además, de esta manera estos valores quedan agrupados en variables, mucho más sencillo de identificar en el código.

El usuario puede también crear tantos ficheros de perfiles como se desee, modificando esos valores, sin la necesidad de cambiarlos del mismo fichero si desea realizar ejecuciones distintas.

Estos atributos, se han recogido como una instancia de la clase **ActorCounters**, para evitar duplicar una clase que resultaría idéntica. Para poder diferenciar por tanto qué instancia se corresponde a los drives actuales del actor y cuál es la instancia donde están los umbrales de los drives, se utiliza un atributo denominado *type*.

Por tanto, una instancia creada para este fin quedaría formada de la siguiente manera:

- **hunger**: nivel de hambre a partir del cual el actor necesita satisfacerlo.

- **thirst**: valor de sed a partir del cual se activa la necesidad del actor de satisfacerla.

- **dirtness**: valor de suciedad por el cual la necesidad del actor por disminuirla se activa.

- **tiredness**: valor del drive de cansancio a partir del cual el actor siente la necesidad de descansar.

- **boredom**: valor de aburrimiento a partir del cual el actor siente la necesidad de divertirse para disminuir el drive.

- **solitude**: valor máximo de soledad que puede soportar el actor sin tener la necesidad de relacionarse con otro actor.

- **wealth**: mínima cantidad de riqueza (dinero), que puede tener el actor sin sentir la necesidad de trabajar o de realizar la actividad correspondiente para conseguir dinero.

- **actor**: instancia del actor al que corresponden los valores de los umbrales.

- **type**: su valor sería la cadena "Max", indicando de esta forma que dicha instancia se corresponde con los valores de los umbrales de los drives.

Como se ha mencionado en el apartado anterior, anteriormente el rango de valores de los drives era de cero a cinco, un rango bastante pequeño. Contando a su vez con el valor de activación de las necesidades, daba el resultado de que casi nunca se encontraban todas satisfechas a la vez, lo que implica que siempre se realizaban acciones para satisfacer estas necesidades, y no otras que no precisaban de precondiciones sobre los drives. Gracias al aumento del rango se pueden ajustar mejor los valores de activación de las necesidades, aunque es totalmente configurable por el usuario. Es posible configurar un cliente con un bajo valor de activación de cansancio, que implique que necesita estar descansando más que cualquier otra necesidad. De esta forma se pueden dar más prioridad a ciertos drives frente a otros.

4.9 Gustos y emociones del actor influidas por objetos y drives

En el universo AI-LIVE, un actor tiene gustos sobre objetos, y es capaz de transmitir ese gusto a otro actor a través de un acto comunicativo. Este acto crea una relación entre ellos, afectando tanto a las emociones de los agentes como a la relación en sí.

El modelo propuesto por (Marta Jiménez, 2008), define el espacio del estado emocional de un individuo como el área definida por los ejes de coordenadas valencia (en inglés valence, cantidad de placer o satisfacción), y activación (arousal en inglés, nivel de actividad). Las salidas de la interacción de los pares conforman un estado emocional. En la siguiente figura se observarán varios ejemplos de estados emocionales.

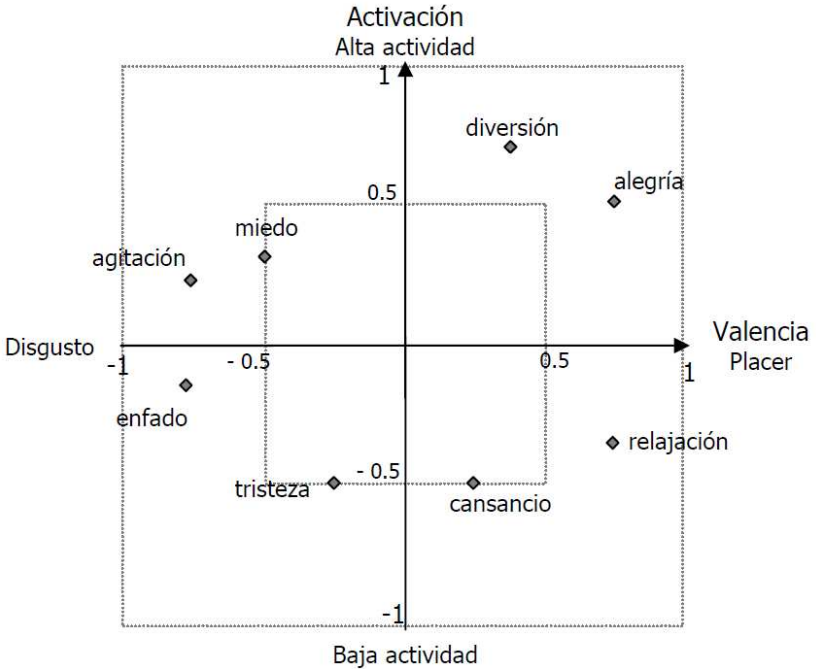

Figura 13: Representación de los valores de valence y arousal en el eje de coordenadas.

En esta figura, se observa que un valor 1 para la valencia indica que el actor está satisfecho, alegre, mientras que -1 indica que está disgustado, apático...
Para un arousal positivo (cuyo valor máximo puede ser 1 como la valencia), indicamos que el actor está atento, con energía, mientras que con valores negativos expresa una actitud pasiva, aletargada, etc.

Anteriormente la relación del actor con los objetos sólo conllevaba implicaciones a la hora de comunicar sus gustos a otro actor. Se pretende aprovechar esto para dar un nuevo enfoque tanto a los propios objetos y objetivos del actor como a sus emociones. Ahora el actor tiene en cuenta los gustos por los objetos a la hora de realizar alguna acción con ellos, utilizando aquel objeto que más le gusta para realizar la acción. Por ejemplo, a un actor le puede gustar un tipo de comida más que al resto de tipos posibles; cuando acuda a comer comerá el tipo de comida que más le guste.

El estado emocional del actor influía a la hora de comunicarse con otro actor, ya que según sus valores de valencia y activación, podía transmitir el gusto por un objeto de una manera u otra. Es posible que un gusto por un objeto o acción que le guste mucho al actor, se transmita con menos entusiasmo si sus emociones son negativas, que si sus emociones son positivas.

También se pretendía que los drives influyeran en el estado emocional del actor. Es decir, cuanto más alto sean los valores de los drives y más urgencia tenga de cubrir sus necesidades, más se verá afectado el estado emocional de manera negativa. Por el contrario, si el actor tiene cubiertas todas sus necesidades y sus drives tienen valores muy bajos, éste se encuentra más contento y por tanto con unos valores de

valencia y activación más positivos. Esta funcionalidad la implementa el servidor de la aplicación [Escudero, 2011].

A continuación se enumeran los cambios realizados en el cliente para incorporar esta funcionalidad. Implican modificaciones tanto en la ontología como en la base de reglas.

Clase `LikenessLists:` Esta clase se ha creado para recoger las distintas listas de gustos por los objetos. Cada una de ellas consta de un multislot de instancias de la clase Likeness, agrupando de esta manera en cada lista los gustos semejantes, con el objetivo de que cada lista atienda a un distinto grado de aceptación de objetos o acciones.

- **dislikenessList**: es un multislot de instancias de gustos, agrupando aquellas acciones, objetos, etc, que disgustan al actor.

- **likenessList**: lista de gustos agrupados por aquellos que más le gustan al actor.

- **neutrallikelist**: lista de gustos sobre objetos o acciones con un valor neutral.

- **inter_dislikelist**: multislot de gustos que se encuadra entre la lista neutral y la lista de "disgusto" o gustos negativos, para que dichos gustos no sean neutrales o se encuentren en los extremos.

- **inter_likelist**: lista con el mismo propósito que la anterior, pero siendo esta la encuadrada en los gustos positivos intermedios.

Los diferentes gustos son personalizables desde el fichero *.profile*, creando así diferentes clientes con características distintas.

Clase `Actor:` Para evitar tener las listas anteriores en la clase **Actor** se implementó la clase anterior. De esta manera, basta con añadir únicamente un atributo con una instancia de la clase LikenessLists:

- **likenessLists**: instancia de la clase LikenessLists, donde se encuentran las distintas listas de gustos del actor.

Regla `markgreater`: esta regla sirve para realizar inicialmente la elección de los gustos del actor concorde a las listas de gustos de éste definidas en su perfil. Realiza la comparación de dos gustos relacionados con una misma clase de objeto (comida, ocio, bebida, etc), marcando como el que más le gusta aquel que tiene una valencia mayor en la relación del gusto.

Las precondiciones de la regla son:

- instancias de dos gustos, cuyo origen de la relación es el mismo.
- El estado emocional que influye en el actor relacionado a cada gusto.
- Comparar que la clase del destino de la relación de los gustos es la misma, para decidir qué objeto gusta más que otro.

El consecuente de la regla deja uno de los dos gustos como “mayor”, simulando de esta manera que al actor le gusta más un determinado objeto de una clase que otro objeto.

En el resto de reglas del cliente (correspondientes a las necesidades básicas), se ha añadido una nueva precondición, la cual necesita una instancia de la clase **Likeness**, y se compara que el actor sea igual al origen de la relación del gusto, y que el objeto sea igual al destino de la relación. La instancia de la clase Likeness recoge un gusto con el atributo *greater* con el valor *yes*, asegurándonos de esta manera que el actor elegirá como primera opción el objeto que más le gusta.

4.10 Script de lanzamiento de la aplicación AI-LIVE

Desde las primeras versiones de AI-LIVE, se implementaron scripts para facilitar el lanzamiento del sistema. Aún así, era necesario abrir un terminal por cada cliente y servidor que se fuera a lanzar, y realizarlo manualmente. Igualmente, por cada ejecución realizada, en caso de querer modificar el perfil, había que ir al fichero que contenía el perfil y modificarlo manualmente.

Se buscaba optimizar y automatizar el lanzamiento de la aplicación AI-LIVE, y se realizó una mejora considerable sobre el script que ya existía.

4.10.1 Script en bash configurable desde línea de comandos

En conjunción con [Escudero, 2011], se desarrolló nuevo script configurable para ejecutar de manera global toda la aplicación. Permite elegir el número de actores que se quieren lanzar con sus perfiles, si se desea lanzar el interfaz gráfico y si se quiere ejecutar un cliente manual. De esta manera se simplifica y acelera la labor del usuario de arranque de toda la aplicación.

4.10.2.1 Fichero de configuración de variables

Primero se creó un fichero de configuración con todas las variables necesarias para arrancar tanto el servidor como los distintos tipos de clientes, así como con las rutas de acceso a los distintos directorios de cada módulo. Primero contiene las siguientes variables generales:

```
#general variables
ontology=ontology.clp   #ontology name (must be at project's home dir
port=6969                     #server port incomming connections
sizex=80                      #geometry x xterm window
sizey=20                      #geometry y xterm window
posx=495                      #posx increments to xterm windows
posy=320                      #posy increments to xterm windows
cleanfiles=./borraFicheros    #script to clean files
host=localhost                #ip/host to connect
```

Las variables generales indican entre otras cosas el nombre del fichero de la ontología, el tamaño de las terminales que se crearán para arrancar cada módulo, el nombre del fichero que borra los ficheros de trazas de ejecuciones anteriores (para servidor y clientes), y la variable host, que indica el host al que conectan los clientes.

A continuación se muestra el apartado de las rutas a los directorios. Estos son el directorio raíz de la aplicación, el directorio en el cual están los archivos correspondientes a los perfiles de distintos clientes, y los directorios de servidor y clientes.

```
#paths
rootdir=$(dirname $(readlink -f $0))      #full path to actual root
directory of project(where this script resides)
profilesdir=${rootdir}/profiles           #profiles folder (under root
directory)

#paths to servers and clients
serverPATH=${rootdir}/server-source       #path to server source dir
clientPATH=${rootdir}/client-source       #path to client source dir
clientManualPATH=${rootdir}/clientmanual-source       #path to client
manual source dir
clientGUIPATH=${rootdir}/client-gui             #path to gui source dir
```

Finalmente se encuentran las variables del servidor y de los clientes. Para ejecutar el servidor es necesario conocer el nombre del script que lo arranca y el fichero con el estado inicial (*initial*.*state*). Para los clientes, además del nombre del ejecutable, se necesita definir el escenario al que se conectará. Estas variables son:

```
#server variables
initial_state=initial.state          #initial state for start
server=./server                      #server executable

#client variables
client=./client                      #client executable
stageC=DEFAULT_STAGE                 #default stage name to connect

#clientmanual variables
clientmanual=./client-manual         #manual client executable
stageM=DEFAULT_STAGE                 #default stage name to connect

#client-gui variables
clientgui=./gui                      #gui executable
```

De cualquier forma, en caso de modificar algún directorio o nombre de archivo, basta con especificar el cambio correspondiente en el script.

4.10.2.2 Fichero con el contenido del script

El script en primer lugar comprueba que existe el fichero de configuración descrito en el apartado anterior, ya que sin él no se pueden leer las variables que contienen las rutas de los directorios ni los ficheros necesarios para ejecutar la aplicación.

```
if [ -f conf-ialive ]
then
      . conf-ialive
else
      echo "configuration file not found"
      echo "\"conf-ialive\" must be on root directory with run.sh"
      exit 1
fi
```

Se ha añadido una función de ayuda, que muestra las opciones disponibles y su significado. Este menú de ayuda aparece tanto si se ejecuta el script sin ningún parámetro, como si se incluye el argumento "–h" o "-?" A continuación se explicará qué ejecuta cada una de las restantes opciones.

```
function help {
 echo "usage: ./run.sh -C [server|manual|client] -s -c [profile] -m
[profile] -g"
      echo "Explanation:"
        echo "-C           -> clean files must specify an option"
      echo "    server      clean server files"
      echo "    client      clean client files"
      echo "    manual      clean client-manual files"
      echo "-c (profile) -> launch   Inteligencia Artificial   client
using the profile given "
      echo "            (without .profile extension!)"
      echo ""
      echo "-m (profile) -> launch manual client using the profile
given"
      echo "                (without .profile extension!)"
      echo ""
      echo "-s       -> launch server"
      echo "-g       -> launch gui client"
      exit 0
      }
```

- La opción "–C" sirve para llamar al script *borraFicheros* (o el que haya indicado en el fichero de configuración), que elimina los ficheros de trazas de ejecuciones anteriores. Por ejemplo, establecer como opciones "-C server –C client –C manual" ejecutará los scripts *borraFicheros* que se encuentren en los directorios *server-source*, *client-source* y *clientmanual-source*.

- La opción "–s" se utiliza para lanzar el servidor. Siempre se debe ejecutar antes que los clientes, y no precisa ningún argumento.

- La opción “–c” sirve para conectar clientes CLIPS. Para lanzar varios clientes basta con incluir la opción “–c” tantas veces como clientes de Inteligencia Artificial queramos. Es necesario a continuación de la opción “–c” añadir el nombre del fichero del perfil (sin la extensión *.profile*) con el que queremos que se conecte el cliente. En caso de no querer lanzar ningún cliente de Inteligencia Artificial basta no poner la opción “–c”.

- La opción “–m” funciona igual que para la opción “–c”, sólo que esta opción es la que lanza los clientes manuales. Se pueden lanzar tantos como se quiera, pero siempre la opción “–m” seguida también del fichero de perfil del cliente. Igualmente si no se desea conectar ningún cliente manual simplemente no hay que activar la opción “–m”.

- Por último, la opción “-g” es la encargada de activar el cliente GUI. En caso de no querer ejecutar la interfaz basta con no añadir esta opción a la hora de ejecutar el script.

El resto del contenido del script realiza la ejecución de cada módulo. Por cada módulo, se accede a su directorio correspondiente, y se lanza su ejecutable en una nueva terminal con los parámetros de entrada necesarios.

```
if [ $# == 0 ]
then
      help
fi
varM=1
var=0
while getopts "C:c:m:gsh?" flag                          #options passed
to scripts
  do
    case "$flag" in
      C)
           case $OPTARG in
         "server")
            echo "Cleaning server files..."
            cd $serverPATH
            $cleanfiles
            cd ../
            ;;
         "client")
            echo "Cleaning client files..."
            cd $clientPATH
            $cleanfiles
            cd ../
            ;;
         "manual")
            echo "Cleaning client-manual files..."
            cd $clientManualPATH
            $cleanfiles
            cd ../
            ;;
      esac
        ;;
      s)
```

```
          echo "Launching server..."
        cd $serverPATH
        xterm -geometry $[$sizex]x$[$sizey]+0+0 -hold -sb -sl 20000 -
e $server ${rootdir}/${ontology} $initial_state $port  &
          cd ..
        sleep 2
       ;;
     c)
          echo "Launching client $OPTARG"

        cd $clientPATH
        auxy=$posy
        auxx=$[$var*$posx]
        echo "va a cargar el profile:"
        fullpath=${profilesdir}/${OPTARG}
        fullpathls=${fullpath}.profile
        echo `ls -l $fullpathls`
        echo "ruta a la ontologia: `ls -l ${rootdir}/${ontology}`"
        echo "ruta actual `pwd`"
        xterm -geometry $[$sizex]x$[$sizey]+$auxx+$auxy -hold -sb -e
$client ${rootdir}/${ontology} $host $port $fullpath $stageC  &
        var=$[$var+1]
          cd ..
        sleep 2
       ;;
     m)
        echo "Launching manual-client $OPTARG"
          cd $clientManualPATH
          auxx=$[$varM*$posx]
        auxy=0
        #fullpath=${rootdir}/${profilesdir}/${OPTARG}
        echo "va a cargar el profile:"
        fullpath=${profilesdir}/${OPTARG}
        fullpathls=${fullpath}.profile
        echo `ls -l $fullpathls`
        echo "ruta a la ontologia: `ls -l ${rootdir}/${ontology}`"
        echo "ruta actual `pwd`"
        xterm -geometry $[$sizex]x$[$sizey]+$auxx+$auxy -hold -sb -e
$clientmanual ${rootdir}/${ontology} $host $port $fullpath $stageM &
        varM=$[$varM+1]
        cd ..
        sleep 2
       ;;
     g)
        echo "Launching client-gui"
        cd $clientGUIPATH
        $cleanfiles
        $clientgui &
     ;;
     h)
          help
       ;;
     \?)
        help
     ;;
     ?)
        help
     ;;
    *)
     # Should not occur
       echo "Unknown error while processing options"
```

```
        ;;
    esac
done
```

5 PRUEBAS Y RESULTADOS

En este apartado se describen las pruebas realizadas para comprobar las funcionalidades implementadas y se analizan los resultados obtenido en cada una.

- **Prueba 1**: funcionalidad de coger y dejar objetos (reglas PickUpAction, PutDownAction, PickUpFromContainer y PutDownInContainer). Se parte de un estado inicial con un actor en el escenario y una lista de objetos cogidos por él y cuyo peso total alcanza el peso máximo que puede soportar. Estos objetos son de comida.

El actor comenzará con un valor de sed alto, teniendo por tanto la necesidad de saciar su sed. Con ello también se comprobará la funcionalidad desarrollada para este nuevo drive. El objetivo de esta prueba es que el actor deje en el contenedor adecuado (refrigerador en este caso), alguno de los objetos que lleva encima para que pueda coger una bebida.

La traza generada de la ejecución ha sido la siguiente:

1. En primer lugar el actor ejecuta la acción `PutDownGoal`, para localizar un refrigerador y dejar un objeto de los que tiene cogidos. En este ejecución el objeto a dejar es `[Fruit_2_id]` en el contenedor `[Refrigerator_1_id]`, situado en la celda (4,18). Se actualiza la instancia de la clase Target para acudir a la celda donde está el refrigerador.
2. El actor se mueve a la celda de acceso de `[Refrigerator_1_id]`, (4, 17), realizando la acción `move`.
3. El actor ejecuta la regla `PutDownInContainer`, depositando el objeto `[Fruit_2_id]`. Este objeto pasa a la lista de objetos que contiene el refrigerador, y el actor ya no lo tiene cogido.
4. Teniendo el actor peso libre, realiza la acción `DrinkObjectGoal`, ya que se cumplen las precondiciones de la regla para satisfacer su sed. Se marca como objetivo una bebida del refrigerador para cogerla, en este caso `[Water_4_id]`.
5. Al estar situado en la misma celda de acceso el actor no se mueve, y ejecuta la regla `PickUpFromContainer`, cogiendo el objeto `[Water_4_id]`, que pasa a la lista de objetos cogidos por el actor.
6. Al querer saciar su sed, realiza la acción `Drink` con el objeto `[Water_4_id]`, disminuyendo el valor del drive correspondiente a la sed.

- **Prueba 2**: funcionalidad de coger y dejar objetos (reglas PickUpAction, PutDownAction, PickUpFromContainer y PutDownInContainer). Se parte de un estado inicial con un actor en el escenario y una lista de objetos cogidos por él y cuyo peso total alcanza el peso máximo que puede soportar. En esta prueba el actor tiene cogido dos objetos de ocio.

El actor comenzará con un valor de sed alto, teniendo por tanto la necesidad de saciar su sed. Con ello también se comprobará la funcionalidad desarrollada para este nuevo drive. El objetivo de esta prueba trata de que el actor al comenzar deje alguno de los objetos que lleva en alguna celda (para esta prueba deberá realizar la acción PutDownAction), aligerando el peso de los objetos que tiene cogidos, para poder coger una bebida.

La traza generada de la ejecución ha sido la siguiente:

1. En primer lugar el actor ejecuta la acción `PutDownAction`, para depositar un objeto en el suelo. Se encuentra en la celda (7, 15) y deja el objeto `[Ball_1_id]` en la celda adyacente (7, 16). Con esta acción se libera peso de los objetos cogidos.
2. El actor decide cambiar de escenario mediante la regla `ChangeStageGoal`. Fija como celda objetivo el acceso a otro escenario (6, 2) para moverse posteriormente.
3. El actor ejecuta la regla `move`.
4. Realiza la propia acción `ChangeStage` para cambiar de escenario.
5. Una vez en el nuevo escenario, el actor localiza objetos de bebida. Como este escenario contiene las tiendas de bebida y comida, realiza la acción `DrinkShopObjectGoal`. Fija de nuevo las coordenadas donde está situada la tienda y se mueve a su celda de acceso.
6. Una vez en la celda de acceso de la tienda, se dispone a comprar bebida `[Wine_10_id]` mediante la acción `BuyDrink`. En este caso se cumplen las precondiciones de que el actor pueda coger objetos debido a su peso, y que tenga el suficiente dinero para comprar dicho objeto.
7. El actor compra otro objeto de bebida: `[Water_10_id]`
8. El actor compra otro objeto de bebida: `[Wine_8_id]`
9. Realiza la acción `Drink` y selecciona el objeto `[Water_10_id]`

- **Prueba 3**: funcionalidad de todas las acciones para cubrir las necesidades básicas de un actor.

Lo que se espera de esta ejecución es una muestra de que el actor realiza toda la variedad de acciones existentes para cubrir sus necesidades básicas, y de observar su comportamiento y fluctuaciones de los valores de los drives a lo largo de los turnos, comprobando también el funcionamiento de aquellas acciones que duran más de un turno.

Al haber un único actor, para satisfacer la necesidad de soledad tendrá que utilizar el teléfono para comunicarse con otro actor y disminuir su drive de soledad.

En la Figura 13 se muestra la gráfica con la variación de los drives a lo largo de toda la ejecución. En el eje X se representa el turno y en el eje Y el valor de cada drive.

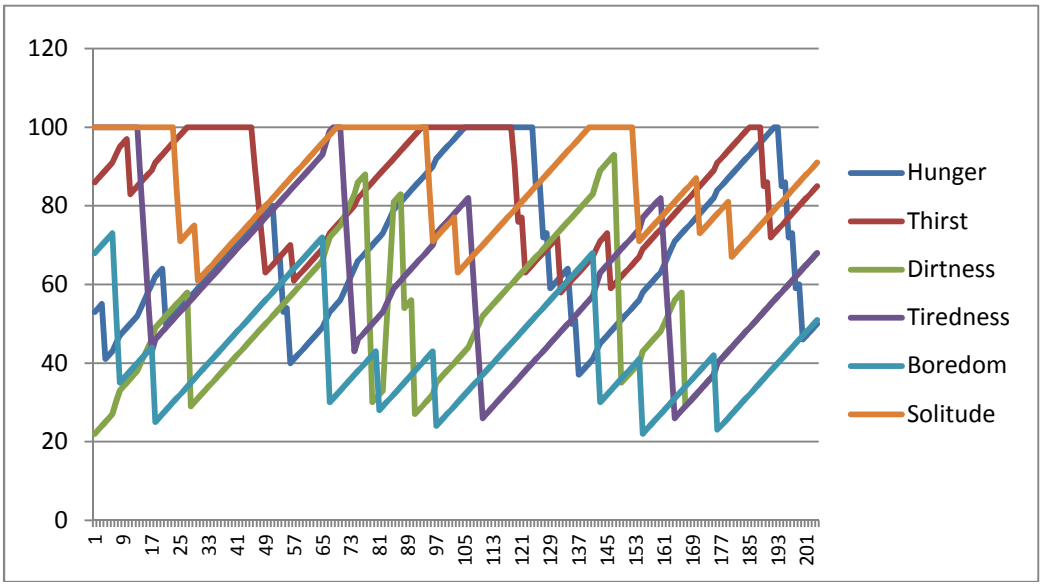

Figura 14: Gráfica de los drives del actor en la prueba 3.

En la gráfica se puede observar el crecimiento de los drives a medida que pasa el tiempo, uno de los objetivos planteados para el proyecto. Se puede observar también que el drive *solitude* aumenta y también desciende, lo que quiere decir que el actor disminuye su soledad cuando tiene la necesidad. Esto lo realiza mediante la acción de comunicación a través de un teléfono.

Se puede observar también de esta ejecución que a pesar de empezar con ciertos valores de los drives muy altos, el actor consigue satisfacer esas necesidades y tener los drives en unos valores intermedios. En este ejemplo, se aprecia la acción de descanso del actor donde disminuye su cansancio y posteriormente a medida que pasa el tiempo vuelve a tener la necesidad de descansar.

El drive *thirst* también asciende y desciende, por lo que se verifica que la implementación del drive y de las acciones para satisfacer la necesidad de la sed están funcionando en el sistema.

Otro aspecto que se puede destacar de la gráfica es el descenso y aumento rápido del drive *dirtness* entre los turnos setenta y noventa. Esto es debido a que el

actor decidió ducharse o bañarse, y posteriormente se dispuso a realizar una actividad de ocio que implicaba un esfuerzo físico, por lo que la suciedad aumentó más rápido que si no hubiera realizado ninguna actividad, donde ascendería lentamente y de manera continuada debido al paso del tiempo. Además, en ese intervalo de turnos coincide con que la pendiente del drive del cansancio aumenta de manera más rápida debido a la realización de la actividad con el esfuerzo físico, provocándole mayor cansancio. Este era otro de los objetivos a alcanzar en el desarrollo del proyecto: la influencia de la utilización de los objetos en los drives, tanto para satisfacer una necesidad básica, como indirectamente aumentar otras.

- **Prueba 4**: integración del modelo emocional al sistema. Con esta prueba tratamos de comprobar que dada una ejecución con más de un actor en el escenario, además de realizar todas las acciones que satisfacen las necesidades básicas, también realiza aquellas acciones implementadas en el modelo emocional [Marta Jiménez, 2008]. Además, cada actor se configuró con valores diferentes de activación de las necesidades básicas.

En las Figura 14 y 15 se muestra la variación de los drives del actor Mike y Amy respectivamente, a lo largo de los turnos de juego.

Actor con perfil Mike:

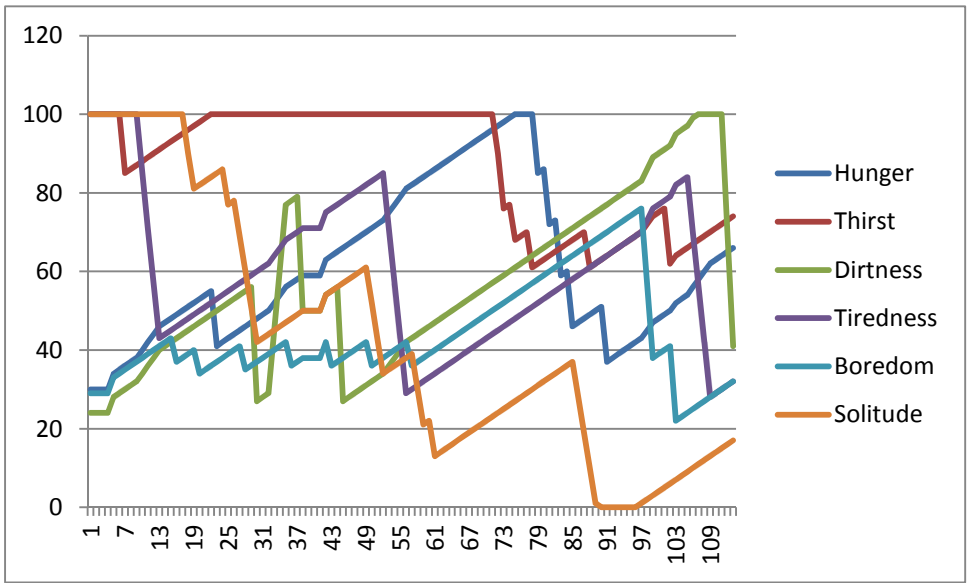

Figura 15: Gráfica de los drives del actor Mike en la prueba 4.

En el perfil de cliente del actor Mike, se asignó un valor bajo de activación del drive soledad lo que hizo que ejecutará la acción de comunicación verbal con el otro actor en repetidas ocasiones. En la gráfica se muestra que el drive de soledad, incluso llegó a estar a cero durante varios turnos. También se puede comprobar que el drive de sed alcanzó el máximo valor durante varios turnos porque el actor prefirió satisfacer otras necesidades antes.

Además, en este perfil venían definidas las siguientes listas de gustos de acciones y objetos:

```
(dislikenessList  Bathtub [Working])
(likenessList Ball Water Meat)
(neutrallikeList Ipod)
(inter_likeList Tv)
(inter_dislikeList Sofa [Washed])
```

A continuación verificaremos a través de las trazas de ejecución que un actor realiza una acción con los objetos que más le gustan.

Como se puede ver, en la lista de gustos positivos del perfil se encuentra una pelota. En las trazas hemos observado que el actor cuando quiere utilizar un objeto de ocio utiliza la pelota, ya que es el objeto que más le gusta dentro de los objetos de ocio:

```
********** regla PickUpAction **********

     - objeto cogido : [Ball_1_id]

     - catched : ()

********** regla play **********

     - objecto de juego seleccionado : [Ball_1_id]
```

En otro momento de la ejecución se ha observado que el actor ha utilizado otro objeto de ocio para disminuir su aburrimiento. Esto ha sido debido a que en esta ejecución sólo existe una pelota en el escenario, que estaba cogida por otro actor. Por tanto decidió coger el siguiente objeto de ocio que más le gusta (en este caso un ipod, que está en la lista neutral de gustos).

```
********** regla play **********

     - objecto de juego seleccionado : [Ipod_1_id]
```

Por otro lado, comprobamos que el sofá se encuentra en la lista de gustos intermedios negativos. Los objetos restantes que no se definen en estas listas se crean con unos valores neutrales cuando el cliente se conecta. Por tanto, dentro de los objetos de descanso, la cama se creará con un valor neutral, y el actor preferirá utilizar una cama para descansar antes que un sofá. Mediante las trazas verificamos que esto se cumple:

```
********** regla RestingObjectGoal**********

     - vamos a por: [Bed_2_id]
     - situado en x :10 y : 4
     - coordenadas actuales del actor: X: 8 Y: 18
     - celda de acceso al objeto : [DEFAULT_STAGE_0_9_2]

********** regla move **********

     - Se va a mover el actor con el Id: [CLIPS_ACTOR_CAEBUNMMOS]
     - Coordenadas futuras del actor: X= 9 Y= 2

     - Vamos a por objeto: [Bed_2_id]
```

```
********** regla Sleep **********

     - objeto de descanso : [Bed_2_id]
```

Para los objetos de comida y bebida se ha obtenido el mismo resultado. El actor en este caso elige comer carne y beber agua (Meat y Water, los tipos de comida y bebida que más le gustan). Cuando se agota la carne o el agua del frigorífico, puede decidir ingerir el siguiente tipo de comida o bebida que le gusta, o acudir a la tienda a comprar.

```
********** regla EatObjectGoal **********

     - vamos a por: [Refrigerator_1_id]
     - y a por: [Meat_2_id]
     - situado en x: 4 y: 18
     - coordenadas actuales del actor: X: 3 Y:12
     - celda de acceso al objeto : [DEFAULT_STAGE_0_4_17]

********** regla move **********

     - Se va a mover el actor con el Id: [CLIPS_ACTOR_CAEBUNMMOS]
     - Coordenadas futuras del actor: X= 4 Y= 17

     - Vamos a por objeto: [Refrigerator_1_id]

     - Vamos a por target: [Meat_2_id]

********** regla PickUpFromContainer **********

     - objeto seleccionado : [Meat_2_id]

     - target seleccionado : [Meat_2_id]

********** regla Eat **********

     - alimento seleccionado : [Meat_2_id]
```

Actor con perfil Amy:

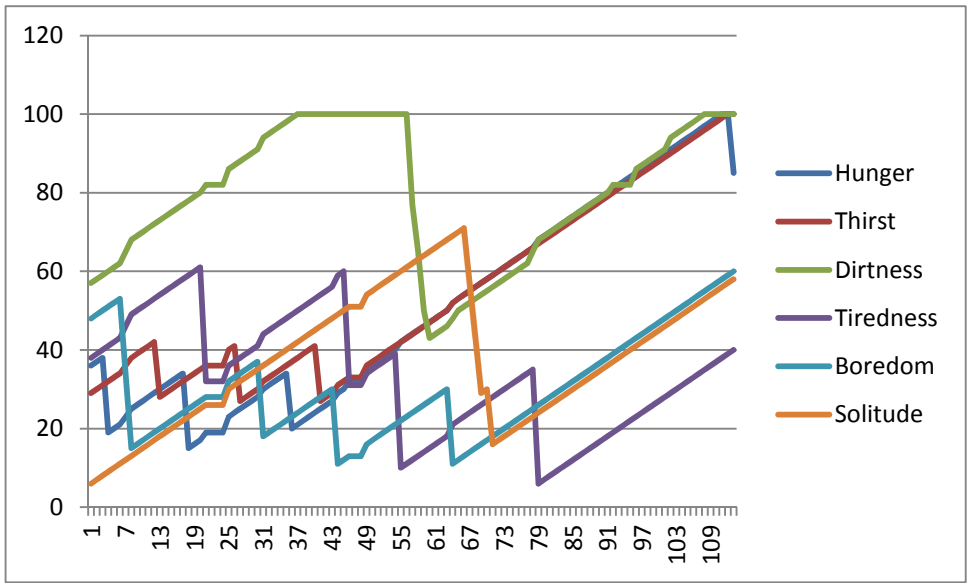

Figura 16: Gráfica de los drives del actor Amy en la prueba 4.

Para este perfil, distinto al del otro actor, se observa también un comportamiento diferente. En este caso los umbrales de los drives andan muy parejos unos de otros y por eso aparecen más agrupados. Destacamos en este caso que el umbral de soledad del actor es distinto al del otro actor. Las acciones comunicativas afectan a ambos actores en su relación y emociones, pero sólo disminuye el nivel de soledad en aquel actor que acude a hablar con otro.

6 PRESUPUESTO

En este apartado se detalla el presupuesto de este proyecto. Este presupuesto engloba todos los costes de personal, los costes materiales y los costes indirectos.

El tiempo que se ha necesitado para la elaboración del proyecto ha sido de 800 horas. Suponemos que cada mes son 20 días laborables.

6.1 Desglose por actividades

Para la elaboración del presente proyecto hemos seguido una serie de fases, en la siguiente tabla se muestran las distintas actividades y el tiempo que ha requerido cada una de las mismas.

TAREAS	***HORAS***
Análisis	100
Diseño	75
Implementación	300
Pruebas	150
Documentación	175

Figura 17: Cuadro de etapas del proyecto

6.2 Costes de personal

Para la elaboración de la aplicación ha sido necesario adoptar roles distintos para poder adaptarse a cada una de las actividades que forman un proyecto. La siguiente tabla muestra el coste total de cada rol utilizado en la elaboración del proyecto. Para el cálculo del presupuesto del personal, se ha tomado el coste hombre/mes de una empresa de soluciones tecnológicas que proporciona desarrollo de software para el sector informático.

Para nuestro proyecto dispondremos de un analista que se encargará de realizar la tarea de análisis del sistema, un ingeniero senior que será el jefe de proyecto y además se encargará del diseño, un programador encargado de la implementación, y dos ingenieros junior, uno realizará las pruebas del sistema y otro realizará toda la documentación necesaria para la elaboración del proyecto.

A continuación se muestra la tabla de los costes de personal del trabajo realizado:

CARGO	*DEDICACIÓN (H/MES)*	*COSTE (H/MES)*	*COSTE TOTAL*
Analista	40	275€	687,5€
Ingeniero Senior	30	600€	1.500€
Programador	50	300€	1.800€
Ingeniero Junior	50	150€	450€
Ingeniero Junior	30	50€	275€

Figura 18: Cuadro de costes de personal

6.3 Costes materiales

En esta sección se detalla los gastos de software y hardware del proyecto. Suponemos la amortización de los equipos a 36 meses.

DESCRIPCIÓN	*COSTE*	*DEDICACIÓN*	*COSTE IMPUTABLE*
Ordenador de desarrollo	899€	650h	215€
Ordenador de pruebas	599€	150h	50€
Sistema Operativo Windows 7 Professional	285€	175h	20€
Openoffice.org	0€	100h	0€
Sistema Operativo Linux (Ubuntu 10.04)	0€	625h	0€
Herramientas desarrollo C, CLIPS, Bash	0€	600h	0€
Microsoft Office 2007	279€	175h	18€

Figura 19: Cuadro de gastos materiales

El total de gastos materiales asciende a 303 €.

6.4 Costes indirectos

Los gastos indirectos se consideran un 20% de los costes totales del proyecto. El coste total del proyecto lo forma la suma de la amortización del coste material y el coste de mano de obra. En total el coste del proyecto es 5.012 euros. El 20% es 1.003 euros. Este es el coste indirecto del proyecto.

NOTA: Cabe destacar que debido a la normativa de la Universidad Carlos III, no se han tenido en cuenta el margen de riesgo ni los beneficios estimados en el cálculo final del presupuesto.

6.5 Resumen del presupuesto

La siguiente tabla muestra un resumen de todos los costes involucrados en el proyecto, así como la suma total de los mismos.

RECURSO	***COSTE TOTAL***
Coste de personal	4.712,5€
Costes materiales	303€
Costes indirectos	1.003€
Total del presupuesto	6.018,5€

Figura 20: Resumen del presupuesto

7 CONCLUSIONES

En el presente apartado se exponen las conclusiones obtenidas tras la realización de este proyecto.

Como primera conclusión destacar que el sistema desarrollado, junto con la integración del modelo emocional de Marta Jiménez, hace de AI-LIVE una aplicación más completa que la existente anteriormente, ya que la variedad de acciones, junto con la influencia de los gustos entre otros aspectos aumenta la riqueza y distintas posibilidades de los actores en cada ejecución, ayudado además por los diferentes perfiles y actores que se pueden crear.

En primer lugar, nos encontramos con un proyecto grande, en el cual han colaborado otros tantos alumnos, donde hay mucha documentación generada por cada uno de ellos, y una persona que se incorpora al proyecto necesita un tiempo de asimilación del funcionamiento de la aplicación, para comprender lo que ya hay desarrollado, y qué se puede desarrollar desde el punto en el que se encuentra la aplicación. Este tiempo de lectura de documentación y de código se ha pretendido reducir generando manuales y distinta documentación, destinada a facilitar a los alumnos que se incorporen posteriormente a entender la aplicación rápidamente para poder realizar sus pertinentes diseños e implementar código.

El desarrollo del proyecto en paralelo con la parte del servidor y con otros compañeros se antojaba complicado por la modificación a la vez de los fuentes del código CLIPS o de la ontología, teniendo que tener precaución para que los cambios en cliente y servidor funcionaran al compilar y ejecutar la aplicación. Optando por utilizar un repositorio de subversión en Google Code, se consigue mayor sincronización, al partir de una versión de la aplicación considerada como estable, y teniendo las distintas modificaciones controladas.

La implementación del actual script de lanzamiento de la aplicación, ha servido para facilitar la utilización al usuario, tanto porque resulta intuitivo mirando el manual o la ayuda del script, como la personalización de los parámetros que se deseen modificar de una manera sencilla. Considero este nuevo script como una importante evolución del anterior, ya que su configuración era más costosa.

Otro de los aspectos es que, anteriormente, era difícil realizar una depuración paso a paso para probar las distintas funcionalidades desarrolladas. Gracias al cliente manual se solventa en cierto modo este problema, ya que permite al usuario elegir la acción a realizar en cada turno (y así comprobar su funcionamiento). Aún así, para la implementación de una nueva regla, este tipo de depuración no es suficiente, ya que cuando una regla no aparece en la lista de acciones a elegir por el usuario, es difícil

deducir a simple vista si la no activación de la regla se debe a la implementación en sí, o a que alguna precondición no se cumple, teniendo que observar la base de hechos.

Gracias al desarrollo del cliente gráfico, tanto por vistosidad como para el desarrollador, se puede observar la información existente en el escenario, así como los valores de los drives y la acción que está realizando el actor actual. Uno de los inconvenientes es la carga del escenario por cada turno, dando la sensación de "teletransportación" del actor de una casilla a otra, sin una representación del actor moviéndose. Otro inconveniente es la utilización de cámaras fijas, en alguna de las cuales para escenarios de tamaños diferentes no se consigue ver una parte clara del escenario.

Por último reseñar, que este proyecto ofrece múltiples ideas para desarrollar y ampliar la aplicación, añadiendo bien aspectos comunes a alguno de los videojuegos de *The Sims*, o añadiendo nuevos conceptos que no se han visto en juegos de estas características. Con esto enlazamos al siguiente apartado del documento, donde se proponen varias líneas futuras de desarrollo.

8 LÍNEAS FUTURAS/TRABAJOS

8.1 Convertir la aplicación en un sistema multiagente en tiempo real

Este concepto englobaría modificar toda la arquitectura del proyecto AI-LIVE, ya que actualmente es un sistema por turnos que va controlando el servidor y dando paso a los clientes que estén en cola conectados.

Este sistema enriquecería mucho el proyecto a largo plazo, ya que permitiría desde contar con tiempo real para medir las acciones (recordemos que las acciones duran un número determinado de turnos) a, lo más interesante, poder realizar acciones en las que intervengan dos o más agentes. Acciones como jugar, en el que se requiera un juego de dos jugadores; bailar, etc. En la situación actual esto no es posible, salvo la acción de hablar con otro actor. Si un actor quisiera jugar con otro a la pelota, el actor jugaría solo, ya que corresponde a su turno, dejaría la pelota al terminar y cuando vuelva a tocar su turno realizaría otra acción. Por lo que esta posibilidad ahora mismo no se puede dar.

8.2 Implementar habilidades en los actores

Una vez implementadas las necesidades básicas y diferentes acciones y objetos para satisfacerlas, se puede implementar un sistema de habilidades en los actores. Al igual que en el juego *Los Sims*, los actores tendrían varios drives correspondientes a habilidades, como pudieran ser mecánica, carisma, cocina, etc. que mejoran a medida que el actor lee libros acerca de una habilidad, o la practican utilizando un objeto adecuado. Esto proporciona a los actores nuevos objetivos y crea distintas personalidades.

8.3 Mejora del cliente gráfico (GUI)

Sería recomendable realizar mejoras en el cliente gráfico, como por ejemplo la visualización de los drives de los actores, ya que en una ejecución a partir de tres clientes los drives de cada actor salen descolocados en el recuadro asignado a la interfaz para mostrar sus valores.

También sería importante mejorar o actualizar los puntos de visión del escenario o cámaras, ya que en escenarios pequeños la visualización no permite ver de una manera muy clara el escenario, por ejemplo algún tipo de cámara dinámica que siguiera al actor.

Por último sería recomendable mejorar el GUI siendo modificado su código para funcionar con las nuevas versiones de CEGUI y Ogre. Estas versiones han cambiado determinadas librerías que provocan que este cliente no compile, al haberse creado nuevas clases y funciones.

8.4 Interactividad utilizando más escenarios

Se podría ampliar el universo del juego implementando más escenarios. Tanto como casas de otros actores, tiendas, sitios donde trabajar, etc. De esta manera se pueden ubicar ciertos objetos relacionados a una clase en un determinado escenario, o aumentar las posibilidades de relación entre actores, por ejemplo, al tener que pedir permiso a un actor para utilizar un objeto que no es suyo, ya que no está en su casa.

8.5 Creación de más objetos o incorporar utilidades a los existentes.

Como ya se ha visto en la ontología, existen determinados objetos que se utilizan para ser representados en el escenario y que no tienen ninguna utilidad (**MiscelaneousObject**). Esto permitiría en un futuro por ejemplo, utilizar el microondas para preparar la comida, o utilizar determinados muebles para guardar ropa u otros objetos, pudiendo de esta forma interactuar con ellos.

9 BIBLIOGRAFÍA

Documentación de Proyectos de Fin de Carrera previos:

- *[PÉREZ, 2006] Proyecto de Fin de Carrera: AI-LIVE.* Miguel Alfonso Pérez Bonomini. Universidad Carlos III de Madrid. 2006.

- *[BENITO, 2007] Proyecto de Fin de Carrera: Necesidades básicas de actores en universos de realidad simulada.* Miguel Benito García.
Universidad Carlos III de Madrid. 2007.

- *[JIMÉNEZ, 2008] Proyecto de Fin de Carrera: Diseño e implementación de un modelo comunicativo emocional para agentes virtuales en el universo AI-LIVE.* Marta Jiménez Matarranz.
Universidad Carlos III de Madrid. 2008.

- *[ESCUDERO, 2011] Proyecto de Fin de Carrera: Ampliación y mejora del universo virtual AI-LIVE.* Javier Escudero Moreno.
Universidad Carlos III de Madrid. 2011.

Bibliografía ordenada alfabéticamente por el apellido del primer autor:

- **D. Borrajo, N. Juristo, V. Martínez y J. Pazos, "Inteligencia Artificial. Métodos y Técnicas", Centro de Estudios Universitarios Ramón Areces, Madrid, 1993**

- **Brownlee, J., 2002. "Finite State Machines (FSM)", http://ai-depot.com/FiniteStateMachines/FSM.html**

- **Buro,M., Call for AI Research in RTS Games, AAAI-04 AI in Games Workshop, San Jose 2004**

- **Castle, L. 1998. "The Making of Blade Runner, Soup to Nuts!" In *Proceedings of the Computer Game Developers' Conference*, Long Beach, CA, 87-97.**

- **Castronova, E. "Synthetic Worlds: The Business and Culture of Online Games", University of Chicago Press, 2005.**

- **Frank, I. 1999. Explanations Count. In *Papers from the AAAI 1999 Spring Symposium on Artificial Intelligence and Computer Games*, Technical Report SS-99-02, 77-80. AAAI Press.**

- **José R. Hilera y Victor J Martinez. 2000. "REDES NEURONALES ARTIFICIALES", Alfaomega. Madrid. España**

- **Huhns, M., Singh, M. P.: Readings in Agents. *Readings in Agents*. Chapter 1, 1-24, 1998.**

- **R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. Artificial Intelligence, 1985.**

- **Laird, J. E. and van Lent, M. 1999. Developing an Artificial Intelligence Engine. In *Proceedings of the Game Developers' Conference*, San Jose, CA, 577-588.**

- **Laird, J. E. 2000. It Knows What You're Going To Do: Adding Anticipation to a Quakebot. In *Papers from the AAAI 2000 Spring Symposium on Artificial Intelligence and Interactive Entertainment*, Technical Report SS-00-02, 41-50. AAAI Press.**

- **Laird, J.E., and van Lent, M., "Interactive Computer Games: Human-Level AI's Killer Application," Proc. Nat'l Conf. A.I., AAAI Press, Menlo Park, Calif., 2000.**

- **Mac Namee, B., Cunningham, P., A Proposal for an Agent Architecture for Proactive Persistent Non Player Characters, Departmental Technical Report TCD-CS-2001-20, Trinity College Dublin, 2001.**

- **F. Martin Mcneill, Ellen Thro, "Fuzzy Logic, A Practical Approach" Editorial AP Professional, 1994.**

- **Norvig, P., and Russell, S.,"Artificial Inteligence: A modern approach", Prentice Hall Series in Artificial Intelligence, 2003 (Second Edition).**

- **Orkin, Jeff – Monolith Productions / M.I.T. Media Lab, Cognitive Machines Group. "Three States and a Plan: The A.I. of F.E.A.R."**

- **Orkin, Jeff - Monolith Productions, Inc. "Agent Architecture Considerations for Real-Time Planning in Games."**

- **Perron, B., Wolf, M. J. P., "The Video Game Theory Reader", Routledge, 2003.**

- **Rabin, S., Common Game AI Techniques. AI Game Programming Wisdom 2, AIWisdom.com, 2003.**

- **E. von Hippel - MIT Sloan Management Review, 2001 - dialnet.unirioja.es**

- **Wooldridge, M. and Jennings, N. R.: *Intelligent agents: Theory and practice*. The Knowledge Engineering Review, 1995.**

Webs:

- **Aigamedev.com. Assaulting F.E.A.R.'s AI: 29 Tricks to Arm Your Game. http://aigamedev.com/reviews/fear-ai**
- **Aigamedev.com. Living with The Sims' AI: 21 Tricks to Adopt for Your Game. http://aigamedev.com/reviews/the-sims-ai**
- **Aigamedev.com. RTS to FPS: Using Strategy AI in Action Games http://aigamedev.com/questions/rts-fps**
- **Aigamedev.com. Teaming Up with Halo's AI: 42 Tricks to Assist Your Game. http://aigamedev.com/reviews/halo-ai**
- **Aigamedev.com. The Crysis of Integrating Next-Gen Animation and AI http://aigamedev.com/reviews/crysis-animation-integration**
- **Allegro. http://alleg.sourceforge.net/index.es.html**
- **Crysisspain.com. CryENGINE 2 http://crysisspain.foroportal.es/foro/viewtopic.php?t=71**
- **Delta3D.org. Delta3d. www.delta3d.org/index.php**
- **Generation5. Cómo empezar con la Inteligencia Artificial. http://www.generation5.org/content/2004/howto02es.asp**
- **Hl2spain.com. F.E.A.R. vs HALF- LIFE 2. http://www.hl2spain.com/articulo.php?ar=14179**
- **Irrlicht. http://irrlicht.sourceforge.net/**
- **Metodologías de la I.A. (Agentes autónomos y Redes neuronales Supervisadas) aplicadas a NPCs (Non player characters), http://www.redcientifica.com/doc/doc200401210112.html**
- **MindForth AI Engine for Robots. http://mind.sourceforge.net/mind4th.html**
- **Multiverse.net http://www.multiverse.net/index.html**
- **ORTS: http://www.cs.ualberta.ca/~mburo/orts/**
- **Weblogs.madridmasd.org. Videojuegos: Grandes retos para los sistemas inteligentes.**

http://weblogs.madrimasd.org/sistemas_inteligentes/archive/2008/04/24/89956.aspx

- **Wikipedia.org. Aprendizaje Automático. http://es.wikipedia.org/wiki/Aprendizaje_Autom%C3%A1tico**

- **Wikipedia.org. Multiverse. http://en.wikipedia.org/wiki/Multiverse**

10 ANEXOS

En este apartado se recogen otros documentos elaborados. En lugar de incluirlos en este documento, se especifican a continuación enlaces a la página web del repositorio donde se alojan estos documentos.

- Manual de Usuario: se recogen las indicaciones y elementos requeridos para instalar, compilar y ejecutar la aplicación.

  http://pruebaailivesvn.googlecode.com/files/Manual_usuario_Uceda.doc

- Manual de Referencia: recoge información detallada sobre los distintos módulos de AI-LIVE para conocimiento y ampliación de la aplicación.

  http://pruebaailivesvn.googlecode.com/files/Manual_referencia_Uceda.doc

- Tutorial de Instalación de Ogre en Linux, para configurar e instalar los elementos necesarios para ejecutar el cliente GUI:

http://pruebaailivesvn.googlecode.com/files/Turorial_Instalacion_Ogre_Linux.pdf